Cooking With Ruby On Rails 4

A Rails Crash Course

Brian P. Hogan

Cooking with Rails 4

A Ruby on Rails Crash Course

Brian P. Hogan

Cooking with Rails 4: A Ruby on Rails Crash Course

Brian P. Hogan

Publication date 2014

Dedication

To the family, for their support.

Acknowledgements

This is the fifth version of this book, which was originally published in 2006 when Rails 1.0 was just released. It's not the most famous guide to getting started with Rails, but it has helped many people become quite successful with the framework. But those people have helped shape this book into what it is today.

My eternal gratitude goes out to Chris Johnson, Chris Warren, Adam Ludwig, Mike Weber, Jon Kinney, Corey Schulz, Josh Peot, Jeff Holland, Austen Ott, Nick LaMuro, Emma Smith, Kevin Gisi, Gary Crabtree, Carl Hoover, Erich Tesky, and the countless others who've worked through previous versions of this book and provided incredibly valuable feedback.

The original Cookbook example was inspired by “Rolling With Ruby on Rails” parts 1 and 2, by Curt Hibbs.

Thanks to Curt Hibbs, Bruce Tate, Zed Shaw, Dave Thomas, Andy Hunt, Sam Ruby, Chad Fowler, José Valim, and all members of the Rails core team for making Ruby and Rails viable options for application development.

Extra special thanks to Andrew Vahey, Andrew Walley, and Mitch Bullard for their feedback on this current edition. They found some pretty big bugs and helped me squash them.

Finally, I thank you for reading this book.

Table of Contents

Changes xi
1. Beta 2.0 - June 30th, 2014 xi
2. Beta 1.6 - May 4th, 2014 xi
3. Beta 1.5 - April 16th, 2014 xi
4. Beta 1.4 - March 27th, 2014 xi
5. Beta 1.3 - March 14th, 2014 xi
6. Beta 1.2 - January 24th, 2014 xi
7. Beta 1.1 - January 12th, 2014 xii
8. Beta 1 xii
Preface xiii
1. What's In This Book xiii
2. Who Should Read This Book xiv
3. What You Need xiv
4. Conventions xiv
5. Where to go for help xv
1. Introduction And Setup 1
1.1. Basic Concepts 2
1.2. Installing Ruby on Rails 6
2. Your First Rails Application 11
2.1. About Our Application 11
2.2. Creating The Application 12
2.3. Creating Some Basic Pages 14
2.4. Creating a Site-wide Layout 18
2.5. Adding Navigation 19
2.6. Sending Data from the Controller to the View 23
2.7. What You Learned 24
3. Adding Recipes 25
3.1. Creating the Recipes interface using Scaffolding 25
3.2. How did we get all that? 32
3.3. What You Learned 32
4. Ruby Basics 35
4.1. History and Philosophy 35
4.2. Interactive Ruby 36
4.3. Numbers, Strings, Variables 37
4.4. Variables 38
4.5. Logic 40
4.6. Methods, Classes, and Objects 41
4.7. Arrays and Hashes 45
4.8. Blocks 46
4.9. Rules of Ruby 47
4.10. What You Learned 48
5. Exploring and Cleaning Up the Scaffolding 51
5.1. Scaffolding and The Rails Way 51
5.2. Cleaning up the Index page 62

5.3. Cleaning up the Show page 64
5.4. Improving our Layout 65
5.5. Updating the Home Page 66
5.6. What You Learned 66
6. Validating User Input 67
6.1. Rails Validations 67
6.2. Unit Tests 68
6.3. Providing Feedback to Users 71
6.4. What You Learned 73
7. Adding Categories 75
7.1. Creating a category model and table 75
7.2. Adding some default records with Rake 76
7.3. Modifying the Recipes table 77
7.4. Creating an Association Between a Recipe and a Category 78
7.5. Adding Categories to the Forms 79
7.6. Displaying Records and Associations 81
7.7. Before Filters 85
7.8. What You Learned 86
8. Security 89
8.1. Using Global Methods In The Application Controller 89
8.2. Securing Destructive Actions 90
8.3. Hiding things people shouldn't see 91
8.4. Limitations of This Authentication Method 91
8.5. What You Learned 92
9. Managing Styles and Other Assets 95
9.1. How Rails Handles Assets 95
9.2. Creating Some Basic Styles 99
9.3. Styling Forms and Errors 103
9.4. What You Learned 104
10. Using Git and Deploying To Heroku 107
10.1. Preparing For Launch 107
10.2. Managing Source Code With Git 109
10.3. Deploying the Application 110
10.4. Building and Deploying a New Feature With Git 111
10.5. What You Learned 113
11. Beyond The Basics 115
11.1. Writing Documentation with RDoc 115
11.2. Working with the Console 116
11.3. Logging 117
11.4. Writing your own SQL statements 117
11.5. What You Learned 118
12. Where To Go Next? 121
12.1. More On Deploying Rails Applications 121
12.2. Development Tools 122
12.3. Books 122
12.4. Online Resources 123

Index 125

List of Figures

1.1. The MVC Pattern 4
2.1. Default Routing 15
2.2. The Welcome To Rails page 16
2.3. Our Cookbook's Home Page so far 24
3.1. The Recipes table 28
5.1. Rails RESTful Routing 54
5.2. Our form fields 57
5.3. How form fields map to the Parameters hash 58
5.4. Form helpers use objects to determine the URL and method 61
5.5. The modified Index page 63
5.6. The Show Page 64
6.1. User Feedback as provided by Rails validations 73
7.1. The Categories table 75
7.2. Recipes table with the category_id column 77
7.3. Recipe Belongs To a Category 79
8.1. Authentication Prompt 91
9.1. Our Layout 100
9.2. Our design so far 102
9.3. Our error page 104
11.1. RDoc output in HTML 115
11.2. Using the Rails Console to work with objects 117

Changes

This book is currently in beta and is undergoing rapid changes.

1. Beta 2.0 - June 30th, 2014

All code and references are updated to Rails 4.1.2. In addition, the MVC diagram is replaced and content throughout is expanded and replaced. The scaffolding explanation is moved after the discussion of Ruby and merged in with the "cleanup" chapter. The validation chapter is moved after the scaffold cleanup now.

2. Beta 1.6 - May 4th, 2014

Happy Star Wars Day.

This release adds more details about MVC to the introductory chapter and replaces the MVC diagram with a new one to make things more clear, based on student feedback.

3. Beta 1.5 - April 16th, 2014

This release fixes several more typographical errors.

4. Beta 1.4 - March 27th, 2014

This release updates the installation steps for Windows to ensure that Rails version 4 is used instead of version 3.2.

It also addresses several more typographical errors.

5. Beta 1.3 - March 14th, 2014

This release adds more information on the scaffold generator and explains the SQlite database. In addition, the section on Heroku is cleaned up to reflect more modern processes.

Also, the section on deploying with Git and Heroku has been expanded, and the explanation of validation and error display has been updated.

6. Beta 1.2 - January 24th, 2014

- Fixed testing section to ensure the tests worked on Rails 4.
- Clarified how to run an individual test

- Clarified how to run the test suite
- Clarified the URL for the recipes interface
- Fixed typo with port number in chapter 3

7. Beta 1.1 - January 12th, 2014

Fixed a number of open errata including some fixes for routes, scaffolding, and some Ruby code examples that were wrong.

8. Beta 1

Initial release of the book for Rails 4

Preface

Get ready to forget everything you know about web application development, because Ruby on Rails will change your world. Web development with Ruby on Rails is one of the fastest ways to build quality applications in a fraction of the time, and all it takes is a little patience and a little time. If you've ever wanted to get started with this technology, this guide is for you.

This simple yet detailed tutorial will guide you through the steps to build a working web application using the Ruby on Rails framework. This guide is meant to expose you to the Rails framework and is not meant to explain all of the concepts in depth. When you’re finished with this tutorial, you will have a better understanding of the basic concepts behind Rails and you should have a basic understanding of how Rails applications are structured and developed.

1. What's In This Book

This guide is meant to be a very brief and quick overview of how Rails works. It's not a comprehensive guide, but you'll learn what Rails is and how it works as you build a small database-driven application.

We'll start out using Rails to create and host some static pages. We'll learn about how to use commands to create files for us and how Rails' built-in web server works.

Then we'll build a complete slice of an application using Rails' scaffolding command, a single command that generates all the code needed to create an app that lets users create, retrieve, update, and delete records. We'll spend time exploring all the code that the scaffolding gives us.

Scaffolding is a great learning tool, but it's not how professionals build things, and so we'll move on to cleaning up the code it creates and then we'll expand our application by adding another table. We'll learn about database relationships and how Rails makes it easy to pull data together.

Then we'll explore validation and unit testing as we make sure that users can't enter bogus data into our application.

After that we'll look at how easy it is to add authentication to our system. We'll keep it simple in this book, by locking the entire application down with a single username and password.

And then we'll dig into how Rails manages images and stylesheets as we create an interface that works on desktop and mobile devices. To do that, we'll look at Rails' support for Sass, a superset of CSS that gives us variables and other awesome features.

Finally we'll explore how we can place this application online for the world to see, and we'll explore how to maintain the changes to our code in a professional manner with the version control software Git.

2. Who Should Read This Book

If you have basic knowledge of HTML, SQL, CSS, and have written database driven applications in another language, this will be the perfect place for you to get started.

If you've never done a database-driven web app before, but you know the fundamentals of programming, you'll still be able to follow this book, but some of the concepts might be a little more confusing for you.

However, this book doesn't cover basic HTML, CSS, and SQL, nor does it cover how to get started writing code. So if you're rusty on those, you'll want to find a good resources such as the materials at http://webplatform.org.

3. What You Need

You'll need a computer running Windows 7 or 8, or a computer running Linux or OSX. You'll need permission to install software on your computer, as well as an active Internet connection. Alternatively, you can use Nitrous.IO[1], a free platform that gives you a Ruby on Rails IDE and Linux terminal right in your web browser. If you choose that route, you'll be able to skip all of the installation steps in the next chapter.

You'll also need a basic text editor. You can use SublimeText[2] which is a commercial editor with a trial period, or any other text editor that you'd write code in. Do not, however, use a word processor like Microsoft Word. That's for writing words, not code.

Finally, we'll use Ruby, Git, and the Ruby on Rails framework, but you'll see how to install all of those things throughout this book.

4. Conventions

As you work through the book, you'll see some conventions.

When you're asked to type a command in the Terminal, it'll look like this:

```
$ rails new cookbook
```

The dollar sign ($) represents the Terminal prompt. You never type that part of the command.

[1] http://nitrous.io/
[2] http://www.sublimetext.com/

Sometimes you'll see snippets of code you'll type out. Those will look like this:

```
name = "Homer"
age = 42
```

Sometimes you'll see output from the screen, which will look like this:

```
heroku login
Enter your Heroku credentials.
Email: brian@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/brian/.ssh/id_rsa.pub
```

This is a hands-on book, so there will be many places where we'll reference files, folders, methods, and classes. The book's formatting should make it clear which file you're expected to edit.

5. Where to go for help

The book's web page contains a link to the book's forum where you can ask questions and provide feedback. This book is frequently updated and you can check the book's version number in the book's footer. When reporting any problems, please be sure to report the version of the book you're reading.

Chapter 1. Introduction And Setup

Ruby on Rails is a framework designed to make building *high-quality, well-tested, stable, and scalable* database-driven web applications easy. It's written in the highly dynamic Ruby programming language.

Ruby was used mostly in Japan until it drew the attention of a developer from 37Signals. The developer, David Heinemeier-Hansson, was working on a new project called Basecamp. He felt very limited by the languages he was using to develop the project and so he started working with Ruby, taking advantage of all of the built-in features of the language.

In July 2004, he released the Rails framework which he extracted from his work on Basecamp. Several versions later, Rails has burst onto the scene and attracted the attention of many development shops and industry leaders including Microsoft, Amazon, Oracle, Boeing, Thoughtworks, Hulu, YellowPages.com, and many others.

Rails continues to evolve, adding features that offer developers robust platform for modern web development.

In the last eight years, Rails has influenced MVC frameworks in nearly every language used for web application development. Principles from Rails have been incorporated into ASP.Net MVC, CakePHP, Django[1], ColdFusion On Wheels, Grails, Play, and many more. However, the dynamic nature of the Ruby language makes the Rails framework stand out from the rest, letting developers be productive by writing much less code.

Rails Scales

Well-written Rails applications have no trouble scaling to tens of thousands of users. Scaling web applications is difficult work and involves intricate knowledge of database design, optimization, network management, caching, and many other factors. Language is rarely the issue. Take Wordpress, for example. Wordpress is a massively popular platform for running a blog. Failure to correctly set Wordpress up with proper caching can result in significant problems, as the database receives too many connections and the site becomes unavailable. Properly configured Wordpress setups handle hundreds of thousands of users each month with no difficulty.

Rails is the same way. While a default Rails application might choke under extreme load, Rails provides the tools you need to build apps that can scale to hundreds of thousands of users. You just have to learn how those tools work. We'll talk about a few of those issues in this book, but don't worry too much about scaling yet.

[1] Rails borrows from Django a lot too

1.1. Basic Concepts

Rails is a framework that ties some of the best practices of web development together into a tight and cohesive bundle. Once you understand the basic concepts of the framework, your productivity will increase rapidly.

MVC Design Pattern

The MVC or Model-View-Controller pattern explicitly splits up your code and separates business logic from presentation and flow control logic. It also provides mechanisms for easy reuse of code. Traditional web applications written in languages like PHP tend to have scripts intermingled with business logic. Developers can avoid this but without a design pattern such as MVC, the process tends to be trial and error. Microsoft's ASP.Net MVC framework actually follows many of the same patterns that Ruby on Rails implements. And Rails implements some patterns found in other MVC frameworks that came before.

The components of MVC

Models contain all business rules and data interaction. All database-related CRUD (Create, Read, Update, and Delete) code belongs in the model as well. If this pattern is followed correctly, you'll never write a select statement anywhere outside of the model. Instead, you will access your data by calling a method on the model.

Views are what your end users will see. They are the web pages in a web application, or the user screens in a desktop application. They should contain very little presentation logic and should be optimized for reuse. Views should never interact directly with models.

Controllers are the glue of the application. Controllers receive user requests, retrieve data from the models, and then send the appropriate view to the user.

Think of the MVC pattern this way: If you went to a nice restaurant, you would ask to see a menu. Your server would bring you a menu. You can think of your server as a controller, and the menu as a view. You decide you want a burger, which you choose from the menu. The server then takes your request back to the kitchen. The server doesn't make your food though. The cooks in the kitchen do that. Once the cooks in the kitchen finish making your burger, your server takes it and brings it back to you. You can think of the kitchen as the models in MVC.

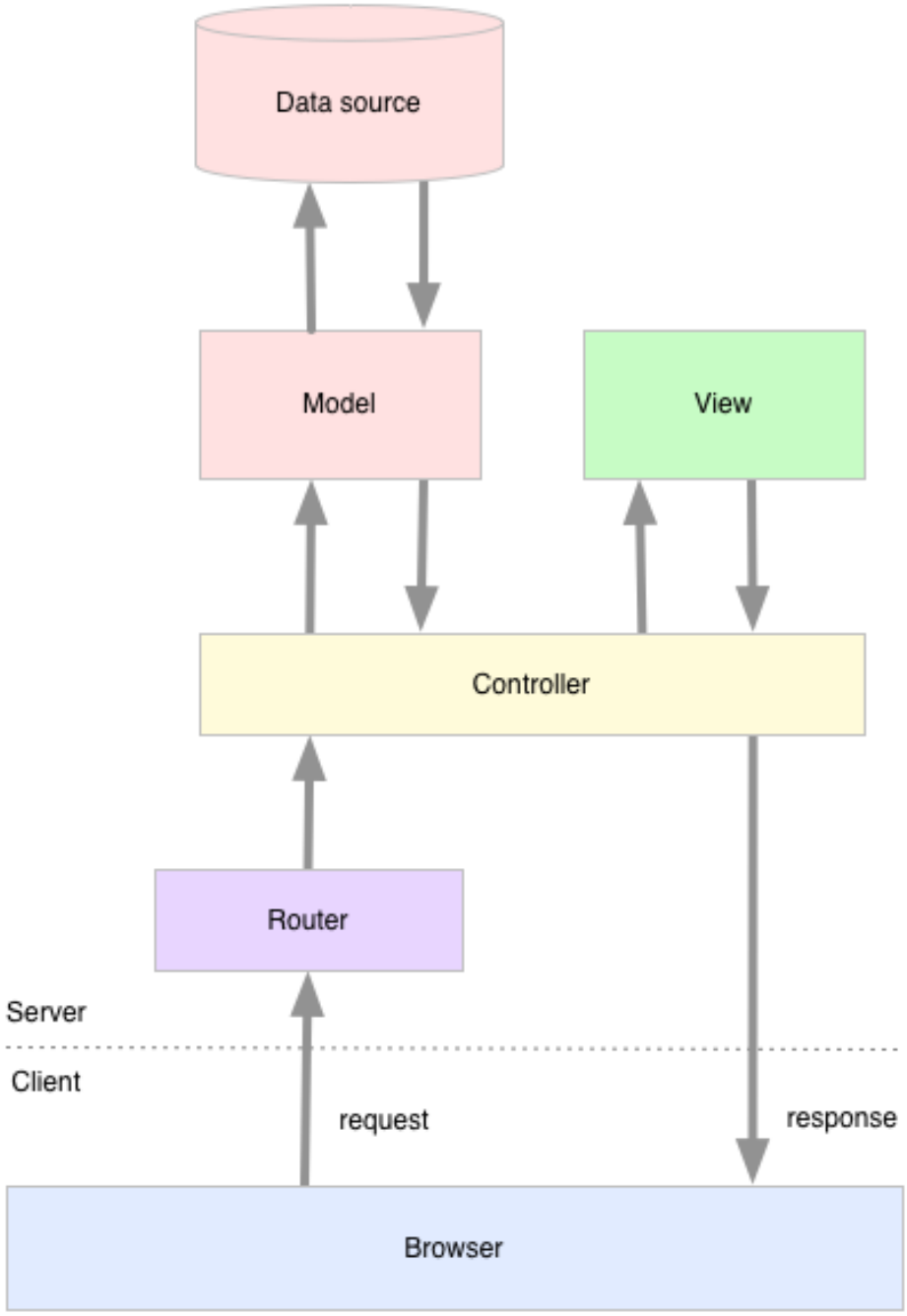
Notice the separation of concerns. You can't talk to the kitchen directly, and the menu doesn't either. Everything goes through your server. That's how MVC works. Models don't talk to views, controllers don't do any business logic, and views don't do anything other than display things to users.

So how does this work in a web application? You make a request through your browser to a certain URL. The server receives that request and the application deter-

mines what controller is associated to that request. It routes the request to that controller. The controller may need some data, so it works with a model to get that data. The controller also needs to construct, or “render” a response, so it grabs a view, which it then sends back as the request. During the rendering process, the data from the model is injected into the view. The response is interpreted by the end user and the user clicks a link, fills out a form, and sends a new request.

The following figure illustrates how Rails implements the MVC pattern:

Figure 1.1. The MVC Pattern



Requests come into the dispatcher which routes to the appropriate controller. Controllers grab data from Models and decide which view to render. The dispatcher then returns the rendered view back to the user in the response.

Rails follows this model very closely, but also adds a few additional components to keep things more organized. As we work through this book, you'll learn more about how Rails implements the MVC pattern.

Asset Management

Modern web applications use a lot of JavaScript files and stylesheets. The Rails framework provides a mechanism to manage these files. In production, your stylesheets are combined into a single file and compressed, reducing download times. JavaScript files are treated the same way.

Rails supports the CoffeeScript and Sass preprocessing languages as well, giving you more power and flexibility.

Test-driven development

Test-driven development (TDD) is a programming methodology where you write tests to drive the design and development of your code. These tests also prove that your code actually works. In an ideal world, you will write your tests first and then write enough application code to make your tests pass.

For example, a developer will create a unit test to design the code that will create a new user in the system. The test will fail until the developer actually writes the code that creates the user. The developer writes the code and continues to run the tests until the tests pass.. If they don’t pass, the developer knows that the functional code is wrong. Once the tests pass, the developer knows they can move on to the next feature. New tests are added when new features are added, and tests are changed to reflect new requirements. Since the tests change with the code, they're always current, and you can use those tests to make sure that you don't break existing features when you add new features.

Rails includes support for unit and integration testing right out of the box. There are no additional libraries or configurations you need to do in order to begin doing test driven development.

Other features and components of Rails

Generator scripts help you create Rails models, views, and controllers.

rails server is a script that launches a Ruby-based web server you'll use while you build your application. This lets you quickly test your application without having to jump through deployment hoops or figure out how to make Rails work with Apache while you're learning how this all works.

Migrations let you define a database-agnostic schema incrementally, with the ability to roll back your changes. You can use Migrations to easily develop on SQLite, stage to MySQL, and deploy on Oracle.

Gems allow developers to add new features to the Ruby language at almost any level. Gems can introduce new features, override existing ones, modify Ruby core

classes, or even share common models and controllers across multiple applications. Gems are easy to write and seamless to use. They allow the Rails core team to deny many feature requests, keeping the Rails framework small and agile.

Rake is a Ruby program that runs user-defined tasks. Rails includes many tasks that help you manage your application, and you can quickly write your own tasks to automate things.

Before we can get going, we'll need to get things installed.

1.2. Installing Ruby on Rails

Rails works on many platforms, but I'll cover Windows, Mac, and Ubuntu here, the three most common platforms. We'll be using Ruby 2.1 and Rails 4.1.2 for this tutorial, as it's available on all platforms.

Quickstart with Nitrous.IO

Nitrous.IO is an online development environment that lets you build and deploy Ruby applications right in your browser. With Nitrous.IO you'll have Ruby, Git, and Rails installed all available, with a simple IDE and a terminal. And it's all free.

To get started in the quickest way possible, visit http://nitrous.io and create a free account. Then create a new development box for Rails. Once your account is confirmed, you'll be able to see your workspace.

In your terminal, issue the command

```
$ cd workspace
```

to move into your workspace. You should always work in this folder on Nitrous.IO. Once that's done you can move on to the next chapter.

Installing on Windows

Installing Rails on Windows can be tricky.

The simplest approach is to download RailsInstaller, a complete toolkit for working with Rails on Windows. [2] Download and run the installer for Windows and accept all of the defaults. At the end of the process, ensure that the option to configure Git and SSH keys is checked.

RailsInstaller will get your environment set up using Ruby 1.9.3 and Rails 3.2. It also installs RubyGems package management tool which we'll use to install some additional Ruby libraries.

[2] `http://www.railsinstaller.com`.

The RailsInstaller also installs Git, a version control system we'll use later in the tutorial, and SQLite3, a simple embedded relational database that we'll use to store our records.

Use Rails Command Prompt

RailsInstaller sets up everything you need to work with Rails on Windows, but you must use it through its special “Rails Command Prompt”. The tools it installs are not available under the normal Windows Command Prompt.

Once the RailsInstaller installation process finishes, you'll be placed at a command prompt which asks for your name and email address. Enter those to configure Git and generate an SSH key which you will use later for deployment.

Finally, since RailsInstaller installs Rails 3.2, run this command to install Rails 4.1:

$ gem install rails -v=4.1.2

And now you're all set. When using Rails, always open the “Command Prompt For Rails”, not the regular command prompt.

Installing on Mac OSX

Lion and Snow Leopard both come with Ruby and Rails installed, but the versions are out of date. We'll use RVM, the Ruby Version Manager, which is a tool that compiles Ruby and all of its prerequisites for our machine. It won't conflict with the Ruby that's installed by default.

Next, you'll need to install XCode from the App Store. It's free, but it's also quite large.

Once XCode is installed you'll need to enable some command line tools that RVM needs in order to install and compile some software. If you're on OSX 10.9, open a Terminal and type the command

```
$ xcode-select --install
```

This will start a small installation program to install additional tools.

If you're on a previous version of OSX, such as 10.8, you can install the Command Line Tools through XCode's preferences menu.

With the compilers installed, open a new Terminal window and type this command:

```
$ \curl -sSL https://get.rvm.io | bash -s stable
```

This will install RVM on your machine into your home directory. When it completes, you'll want to close Terminal and launch it again. Then run this command to install Ruby.

rvm install ruby 1.9.3

The process will take some time to install. Once it completes, you'll want to make this version of Ruby the default for this tutorial:

rvm use --default 1.9.3

Finally, you can install the Rails framework using the `gem` command from the Terminal:

```
$ gem install rails
```

This installs the latest version of Rails and the corresponding dependencies.

Installing Git

Grab the Git Installer for OSX[3] and install it.

Installing on Ubuntu

There are a lot of ways to install Ruby on Rails on Ubuntu, but we'll use RVM, the Ruby Version Manager to simplify the installation. These instructions will be similar to the way you'd set up Ruby on OSX.

First, you'll need to install some compilers and libraries that RVM will use to compile Ruby from source. Open a Terminal session and enter this command:

```
$ sudo apt-get install build-essential curl bison
openssl libreadline6 \ libreadline6-dev curl git-core
zlib1g zlib1g-dev libssl-dev \ libyaml-dev libsqlite3-0
libsqlite3-dev sqlite3 libxml2-dev \ libxslt-dev auto-
conf
```

You'll be prompted to enter your password so the packages can be installed.

With the compilers installed, open a new Terminal window and type this command:

```
$ \curl -sSL https://get.rvm.io | bash -s stable
```

This will install RVM on your machine into your home directory. When it completes, you'll want to close Terminal and launch it again. Then run this command to install Ruby.

[3] `http://code.google.com/p/git-osx-installer/downloads/detail?name=git-1.7.9.4-intel-universal-snow-leopard.dmg&can=2&q=`

rvm install ruby 1.9.3

The process will take some time to install. Once it completes, you'll want to make this version of Ruby the default for this tutorial:

rvm use --default 1.9.3

Finally, you can install the Rails framework using the `gem` command from the Terminal:

```
$ gem install rails
```

This installs the latest version of Rails and the corresponding dependencies.

Installing Git

Use the `apt` package manager to install Git

```
$ sudo apt-get install git-core
```

This installs Git and its related libraries.

> **Working with Older Versions of Rails**
>
> This document is written with Rails 4.0.1 in mind. It is possible that the version of Rails that's currently provided by Rubygems is newer. This may cause some problems as Rails is a constantly-evolving framework. However, it's easy to have multiple versions of the Rails framework installed on your machine, thanks to Rubygems.
>
> The command
>
> ```
> $ gem list rails
> ```
>
> will quickly tell you which versions of Rails you have installed. To install a specific version of Rails, you simply issue this command:
>
> ```
> $ gem install rails -v=4.0.1
> ```
>
> To install Rails 3.2 in order to follow along with some older books and tutorials, install it with
>
> ```
> $ gem install rails -v=3.2
> ```

That takes care of the installation. You're ready to go!

What You Learned

In this chapter you got a brief overview of the various components of Rails and you got your development environment configured.

Now that your machine is configured to work with Ruby on Rails, let's start exploring!

Chapter 2. Your First Rails Application

There's no better way to learn something than by doing, so let's use Ruby on Rails to build a simple database-driven application. Throughout this book we'll put together a simple web-based cookbook that will let us keep track of our favorite recipes. Each recipe will have ingredients, instructions, and a title. Eventually you'll be able to associate a category to the recipe. When we're done we'll be able to look at this on our computers as well as our phones.

In this chapter we'll start with the very basics. We'll learn about our application a bit, and we'll put together the home page of our application as well as a couple of additional pages, and we'll get a chance to see how Rails integrates with HTML.

2.1. About Our Application

We're going to develop a very basic online cookbook application. When it comes down to deciding on the features of an application, it helps to think about them in terms of “user stories”, an approach that asks developers to look at the features of the product from the user's perspective.

A “user story” often follows this format:

- As a _____
- I want to _____
- So that I can _____

We need to be a able to fill in these blanks with a type of user, a specific task, and a reason for doing that task. This helps us stay on track and keeps us focused on adding real value for the users.

Here are the user stories for the application we're going to build:

- As a visitor, I want to see the number of recipes on the home page of the site so I know the site is being used.
- As a visitor, I want to learn about the site so I can decide if the content has any value to me.
- As a visitor, I want to be able to view recipes so I can make something good for dinner.
- As a site owner, I want to be able to create new recipes so I can share them with the world.
- As a site owner, I want to be able to modify existing recipes so I can fix my mistakes

- As a site owner, I want to be able to delete recipes so I can remove junk I don't want to share anymore.
- As a site owner, I want to make sure that the title, instructions, and ingredients for a recipe are required fields so that a recipe is complete.
- As a site owner, I want to be able to place recipes in categories so I can organize them better.
- As a site owner, I want to have to log in to the site be able to create, update, or delete recipes, so that the general public can't modify my database and mess with my records.
- As a visitor to the site, I want to be able to view the site on my mobile phone as well as my desktop.

As we work through this book, we'll address each of these stories. When we're done we'll have a fully working app that includes all of these features. Let's get started!

2.2. Creating The Application

Create a new Rails project by opening a Terminal window and typing

```
$ rails new cookbook
```

This creates a new folder called `cookbook` and generates a bunch of subfolders and files that make up the Rails framework itself. When you run the command you'll see output like this:

```
create
create  README.rdoc
create  Rakefile
create  config.ru
create  .gitignore
create  Gemfile
create  app
create  app/assets/javascripts/application.js
create  app/assets/stylesheets/application.css
create  app/controllers/application_controller.rb
create  app/helpers/application_helper.rb
create  app/views/layouts/application.html.erb
create  app/assets/images/.keep
create  app/mailers/.keep
create  app/models/.keep
create  app/controllers/concerns/.keep
create  app/models/concerns/.keep
...

Using sqlite3 (1.3.9)
Using turbolinks (2.2.2)
Using uglifier (2.5.1)
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
   run  bundle exec spring binstub --all

* bin/rake: spring inserted
* bin/rails: spring inserted
```

This command creates the folder `cookbook` and places a whole bunch of framework files in that folder. It also runs a few additional tasks, including downloading some additional dependencies this new app will need.

This is just one of Rails' generators. As you move through this tutorial, you'll notice that Rails has many generators that build files for you. Generators are great for file creation and laying down code that's generic.

Before moving on, switch directories into this new `cookbook` folder that the `rails` command created for you.

```
$ cd cookbook
```

You'll issue all of your future commands in this folder.

Generating a project using a specific Rails version

You can have multiple versions of Rails installed via RubyGems. And you can choose which version of Rails you wish to use for your application. For example, if you need to use Rails 4.1.2 to follow along with this book, you can install Rails 4.1.2 with

```
$ gem install rails -v=4.1.2
```

and then create the cookbook project with

```
$ rails _4.1.2_ cookbook
```

The `_4.1.2_` parameter tells RubyGems which version of the `rails` executable to use.

A Quick Look at a Rails Application Structure

We just ran a single command that created a huge number of folders and files. A Rails application has a standard structure and each of these folders and files has a purpose.

The `app` folder is where most of your application's code will go. Within the `app` folder there's a folder for `models`, one for `controllers`, and one for `views`. There's also a folder called `helpers` which will contain methods we can use in our views. Finally, the folder called `assets` is where we'll store our stylesheets, JavaScript files, and images. You'll spend most of your time working with files in this folder.

The `config` holds most of the configuration files for our application.

The `db` folder contains files that configure and manage our database schema. As you'll soon learn, you'll be able to use Ruby code to define and manage your database tables, so you can keep your database development in sync with your application development.

The `lib` folder is where you might put your own custom extensions to Rails. It's also where you'll store any automated tasks you wish to write for your application.

The `log` folder contains log files for our application server.

The `test` folder is where all of the model, controller, and application tests go. Test-driven development is an important part of building professional Rails applications, and there's a special place already created for us to do that.

The `tmp` folder is where Rails stores its temporary files.

The `vendor` folder is where any third-party libraries will be installed. This folder isn't used nearly as much now as it was in older versions of Rails.

As we move through the book, you'll get a chance to look at these folders in more detail. But one of the neatest things about Rails is that each application has this exact structure, so you almost always know where to find things. Models will be in the `app/models` folder, and the file that contains your database connection information will be in the `config` folder.

Gems and Bundles

Rails applications use an application called Bundler to manage dependencies distributed as Gems, and your new Rails application already has a several Gems it needs to install, including the Rails framework itself and the *sqlite3-ruby* gem, which lets Ruby talk to the SQLite3 database we're going to use. When we generate a Rails application, the script creates a file called `Gemfile` for us which lists all of the gems we'll need to run our application. It also automatically runs the command

```
$ bundle install
```

which installs all of the gems and their dependencies that are listed in this file. Throughout this tutorial, we'll add additional libraries to the `Gemfile`, and we'll use the command `bundle install` to go and fetch these new dependencies when we do that.

This process creates a file called `Gemfile.lock` that lists the specific version numbers of all Gems the app needs. When Rails starts up, it uses Bundler to ensure that you have those versions installed. And even though you may have already installed a Gem on your system, Rails won't see it unless it's in the `Gemfile`. This makes it possible for you to use different versions of Rails on your various projects without resorting to other forms of Gem management. However, this only works for Rails 3 and above. Previous versions of Rails require more arcane ways of managing dependencies.

2.3. Creating Some Basic Pages

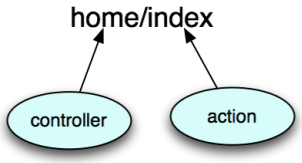
The first two stories on or list of stories talk about some basic information pages:

- As a visitor, I want to see the number of recipes on the home page of the site so I know the site is being used.

- As a visitor, I want to learn about the site so I can decide if the content has any value to me.

We can create these pages relatively easily. In a Rails application, requests are forwarded to controllers and actions based on the contents our the URL:

Figure 2.1. Default Routing



A URL like the above would, by default, look for a "home" controller and an "index" action.

We generate our home page and our about page with a Rails generator. Generators are scripts that create files for you, and you'll use them a lot as you develop applications.

At the terminal, inside of your `cookbook` folder, type this command:

```
$ rails generate controller home index about
```

You'll see the following output:

```
create  app/controllers/home_controller.rb
 route  get "home/about"
 route  get "home/index"
invoke  erb
create    app/views/home
create    app/views/home/index.html.erb
create    app/views/home/about.html.erb
invoke  test_unit
create    test/controllers/home_controller_test.rb
invoke  helper
create    app/helpers/home_helper.rb
invoke    test_unit
create      test/helpers/home_helper_test.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/home.js.coffee
invoke    scss
create      app/assets/stylesheets/home.css.scss
```

This command generates our `Home` controller and two views, or pages called, `index` and `about`. If you wanted to create additional pages, you'd specify additional page names in the generator command. You can also create additional pages later by hand, which we'll do shortly.

Starting The Built-In Development Server

You don't need Apache or a dedicated server to see your Rails application in action. Each Rails application can be launched using a small application server that you launch right in your project folder. Type

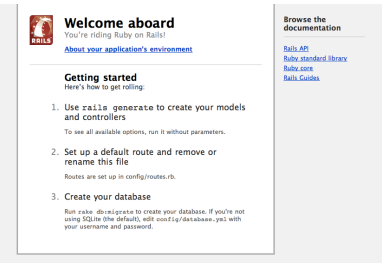
```
$ rails server
```

and you'll see the web server start up on port 3000.

```
=> Booting WEBrick
=> Rails 4.0.1 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-03-29 16:56:54] INFO  WEBrick 1.3.1
[2012-03-29 16:56:54] INFO  ruby 1.9.3 (2011-10-30) [x86_64-darwin11.2.0]
[2012-03-29 16:56:54] INFO  WEBrick::HTTPServer#start: pid=25961 port=3000
```

Open your web browser and go to http://localhost:3000 and you'll see the "Welcome to Rails" page, as shown in Figure 2.2, “The Welcome To Rails page” on page 16.

Figure 2.2. The Welcome To Rails page



The root of a Rails development site shows this default page.

Take a look back at your Terminal window that's running the Rails server and you'll see this output:

```
Started GET "/" for 127.0.0.1 at 2014-01-08 21:21:57 -0600
Processing by Rails::WelcomeController#index as HTML
  Rendered /Users/brianhogan/.rvm/gems/ruby-1.9.3-p385/gems/railties-4.0.1/lib/
rails/templates/rails/welcome/index.html.erb (3.9ms)
Completed 200 OK in 66ms (Views: 19.7ms | ActiveRecord: 0.0ms)
```

This is a log of the request you just made to the application. It shows that you made a request to the root of the site, and it shows how Rails responded to that request.

Now, when we visit the root of the site, we don't see the “home” page we generated. We see the standard welcome page that every new Rails application shows by default. To see your homepage you created with the generator, visit

http://localhost:3000/home/index

Take another look at the output from the web server:

```
Started GET "/home/index" for 127.0.0.1 at 2014-01-08 21:28:58 -0600
Processing by HomeController#index as HTML
  Rendered home/index.html.erb within layouts/application (4.7ms)
Completed 200 OK in 1245ms (Views: 1197.2ms | ActiveRecord: 0.0ms)
```

This shows the new request, and you can see that the request was handled by `HomeController#index`.

You'll learn how to replace the default Rails home page with your custom one at the end of this chapter. For now, let's customize our new pages a bit.

Use Multiple Terminals

Right now, your Terminal is running the development web server, which means you can't run any other generators or commands. Instead of stopping the server, open a new Terminal window and navigate to your `cookbook` folder. Then you can leave the original Terminal running in the background.

Customizing Your Views

The views the generator creates need some work before we can really show them to anyone. In a Rails application, your view files are located in the `app` folder, under the `views` folder. Each controller's views are stored in a folder with the corresponding name. Our view for the `/home/index` action is located at `app/views/home/index.html.erb`. Open that file now, delete its contents, and paste this in instead:

Example 2.1. code/01_static/cookbook/app/views/home/index.html.erb

```
<h2>Latest Recipes</h2>
<p>There are 25 recipes total.</p>
```

We're going to show the latest recipes on our home page. Refresh http://localhost:3000/home/index in your browser to see your change. You can make changes to your views and see them right away without having to restart the development server.

Change the About page

Let's make some changes to the `about` page. Open `app/views/home/about.html.erb` and change its contents to the following:

Example 2.2. code/01_static/cookbook/app/views/home/about.html.erb

```
<h2>About</h2>
<p>This is a simple cookbook application I wrote. I hope you like it.</p>
```

If you visit http://localhost:3000/home/about, you'll see your changes. Of course, typing in these URLs isn't going to be very useful, so let's build some navigation for our cookbook. To do that, we'll need to learn about layouts.

2.4. Creating a Site-wide Layout

You've probably noticed that our view pages have not included any required HTML elements. That's because Rails defines a layout that “wraps” our views with the standard HTML5 template, and we can modify that layout to include whatever HTML we want. We can create a header, footer, add images, and even attach new stylesheets and scripts.

Let's explore the layout system by creating a very simple site-wide layout. We'll modify the default layout file called `application.html.erb` which is located in the `app/views/layouts` folder. By default, it loads a stylesheet, a JavaScript library, and a block of code that emits the individual template pages. The individual view is injected into the layout where you see the keyword

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

The default layout that Rails gives us is very plain, so let's add a little bit of additional markup by using some HTML5 sectioning elements. Change the file so it looks like this code:

Example 2.3. code/01_static/cookbook/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cookbook</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="container">
      <header>
        <h1>Our Cookbook</h1>
      </header>
      <section class="content">
        <%= yield %>
      </section>
      <footer>
        <h4>Copyright <%=Date.today.year %> OurCookbook</h4>
      </footer>
    </div>
  </body>
</html>
```

With these elements in place, we'll be able to identify the various regions of the page. This'll come in handy later when we add styles, but it's also a good practice because it'll improve the site's accessibility.

Every controller will default to using this layout as long as there's not a more specific layout file defined elsewhere. You could, in theory, have multiple layouts, but we'll keep things simple for now.

Dynamic Content with ERb

The layout file you just created has a couple of occurrences of ERb, or Embedded Ruby. We use ERb to mix Ruby code and HTML code together. Code placed between the <% and %> characters is evaluated. This is very similar to ASP, JSP, or PHP.

In the template we just created, we use ERb to call the `yield` method which injects the template into the view.[1]

We're also using ERb in the footer of our layout to display the current copyright year. We use the `today` method on the `Date` class to get today's date as a Date object, and we call the `year` method which returns just the year.

When you use <%= %> (note the equals sign), you're saying you want to print the result on the page. If you leave off the equals sign, the result will be suppressed. There are many occasions where you'll use ERb without the equals sign, including `if` statements and `for` loops.

2.5. Adding Navigation

Our cookbook isn't going to work very well unless we spend some time making some navigation that can help our users get around. We could add navigation to each page we create directly, but instead, we'll just add navigation code to the layout. Open `app/views/layouts/application.html.erb` and add this code above the <%=yield %> section:

Example 2.4. code/01_static/cookbook/app/views/layouts/application.html.erb

```
<nav>
  <%=link_to "Home", :controller=>"home", :action=>"index" %>
  <%=link_to "About", :controller=>"home", :action=>"about" %>
</nav>
```

This is the `link_to` helper. It's used to create hyperlinks between sections of your application. It takes two parameters - the text of the link that the user will see, like "Home" or "About", and a destination for the link. In our above example, it looks like we're passing three parameters - the text for the link, a controller, and an action. That's where things get a little tricky.

Whenever you see the => character in Ruby code, you're dealing with a `Hash`. So, the `link_to` method here is actually taking in two parameters - a string and a hash. The `link_to` method uses the hash to figure out where the link should go. This is the simplest way to use `link_to` in your code, because you can always rely on the knowledge that every request to a Rails application maps to a controller and an action.

[1] `yield` does a lot more than that, but the complex description is not appropriate here.

Routing

Rails has a mechanism called Routing that maps up requests with controllers and actions. When we create controllers and actions, we need to let Rails know how to handle them. Since we create our controller with the `generate` command, some routes were added for us. Open up `config/routes.rb` and you'll see these lines:

```
get "home/index"
get "home/about"
```

These lines define how Rails knows which controller and action a request belongs to. This is a simple default, and we don't want to have to go to `/home/index` just to get to our home page. So let's use Rails' routing features to change how our URLs look.

The Root Route

When we go to http://localhost:3000/, we don't see our home page. We see the generic "Welcome To Rails" page as shown in Figure 2.2, “The Welcome To Rails page” on page 16.

In `config/routes.rb`, locate this line:

```
get "home/index"
```

Remove this line, and change it to this:

Example 2.5. code/01_static/cookbook/config/routes.rb

```
root :controller => "home", :action => "index"
```

Now visiting http://localhost:3000/ shows you your home page, complete with navigation.

Named Routes

While the `:controller => "home", :action => "index"` syntax for creating links is easy to follow, it's also a little verbose. We can make it less so by creating our own routes. Let's make a named route for the "About" page.

Open `config/routes.rb` and remove this line:

```
get "home/about"
```

Then add this line right above the `root` route we previously defined:

Example 2.6. code/01_static/cookbook/config/routes.rb

```
get "/about", :controller => "home", :action=>"about", :as => "about"
```

A named route lets us map any URL we want to any controller and any action. Right now we're mapping `/about` to the `home` controller's `about` action. That means you can visit http://localhost:3000/about now instead of http://localhost:3000/home/about. In fact, the original route no longer responds now that we've redefined it..

We now have a root route and a named route, and these not only help us define what the URL looks like, but they also provide us handy shortcut methods for referencing these routes in our `link_to` calls. Open up `app/views/layouts/application.html.erb` and change the link for the home page from

```
<%=link_to "Home", :controller=>"home", :action=>"index" %>
```

to

Example 2.7. code/02_navbar/cookbook/app/views/layouts/application.html.erb

```
<%=link_to "Home", root_url %>
```

The `root_url` comes from the route name. In `routes.rb` you defined `root :to`. That generates a method called `root_url` that returns the URL to that page.So instead of having to use the hash syntax, you can use this helper.

Every route you declare gets a function that you can use to quickly build links like this. To see the routes and their mappings quickly, you can run

```
$ bundle exec rake routes
```

from the root of your Rails application at the command prompt or terminal. This generates a simple table that shows the prefix, the URL, and the controller and action it points to that looks like this:

```
Prefix Verb URI Pattern      Controller#Action
about  GET  /about(.:format) home#about
 root  GET  /                home#index
```

You just have to add the `_url` suffix when you use it. The `_url` suffix generates a full URL with the protocol, port, and path. The `_path` suffix creates just the relative path.

Rake

Rake is a tool we use with Ruby projects. It allows us to use Ruby code to define “tasks” that can be run in the command line. Rails has a bunch of Rake tasks built-in.

We're prefixing the `rake` command with `bundle exec` because we need to ensure that we run the version of Rake specified in our Gemfile. The `rails` command automatically uses the Gemfile, but Rake doesn't.

So, you can do the same thing with your `about` route you created. The routing table shows that there is an `about` prefix, so we can use `about_url` to generate the URL in our view. Change

```
<%=link_to "About", :controller=>"home", :action=>"about" %>
```

to

Example 2.8. code/02_navbar/cookbook/app/views/layouts/application.html.erb

```
<%=link_to "About", about_url %>
```

The alternate syntax for named routes looks like this:

```
get "/about", :to => "home#about"
```

where the controller and action name are separated by the pound sign. This is the more popular way to define named routes, but there's no difference in the result.

We can also be specific about the route's name. Using the simplified syntax, Rails would figure out that we want to use `about_url` and `about_path` for our helper, but if you want to be explicit about it, you'd use this syntax:

```
get "/about", :to => "home#about", :as => "about"
```

So, let's change both routes to use the the more standard syntax instead of the hash syntax:

Example 2.9. code/02_navbar/cookbook/config/routes.rb

```
get "/about", :to => "home#about", :as => "about"

root :to => "home#index"
```

Now, verify that your links to your pages all still work. Make sure your web server is running. If you've stopped it, make sure you're in a Terminal or Command Prompt and you're in your `cookbook` folder. Then start the server with

```
$ rails server
```

or the more concise

```
$ rails s
```

Then visit http://localhost:3000 and verify that your navigation links still work as expected.

Now that we've explored the basics of creating pages and managing links and URLs in Rails, let's look at how we can work with simple data.

2.6. Sending Data from the Controller to the View

Our `index` view is eventually going to show us how many recipes we have in our database. Right now, the number 25 is hard-coded in the view. We don't have a database yet, but we can still remove that hardcoded number from our view and replace it with Ruby code.

In the MVC pattern, we want to keep as much logic out of the the views as possible. to get the number of recipes, and the most logical place for us to do so is in the controller. Open up the file `app/controllers/home_controller.rb` and change the `index` action so it looks like this:

Example 2.10. code/02_navbar/cookbook/app/controllers/home_controller.rb

```
def index
  @number_of_recipes = 5
end
```

Controllers are Ruby classes, and actions are methods of that class. Rails automatically renders the view file associated with the action, so in this case the `index.htm.erb` view is going to be rendered. The variable we just set, `@number_of_recipes` is an instance variable. In Rails applications, any instance variables we create in a controller are automatically usable in any view, even in the layout.

Open up `app/views/home/index.html.erb` and change this line:

Example 2.11. code/01_static/cookbook/app/views/home/index.html.erb

```
<h2>Latest Recipes</h2>
<p>There are 25 recipes total.</p>
```

to this:

Example 2.12. code/02_navbar/cookbook/app/views/home/index.html.erb

```
<h2>Latest Recipes</h2>
<p>There are <%=@number_of_recipes %> recipes total.</p>
```

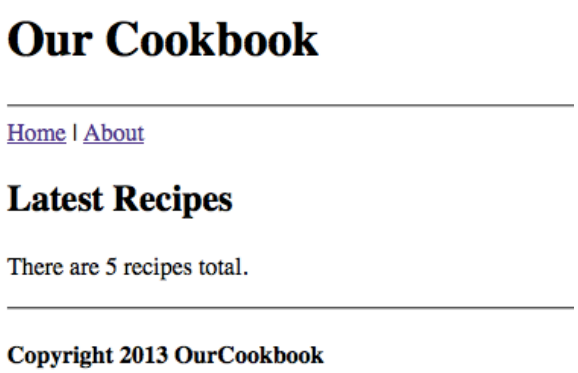
Eventually, we'll replace the `@number_of_recipes` to some code that queries from our actual database. In order to do that, we need to learn how Rails works with databases, and get some recipes into our system.

A little more on Layouts

Layouts work a little differently than you might expect. They can see any instance variables (those ones prefixed with @) set in the controller, including variables created within the view files themselves, because it is read last before being sent to the browser. This means you can set the page title in the individual controller actions as `@title = "some-`

thing" and use it in the layout as `<title><%= @title %></title>`.

Figure 2.3. Our Cookbook's Home Page so far



Our cookbook's home page with layout and navigation in place.'

2.7. What You Learned

So far, you've learned how to create a simple static web site using Rails. You learned that controllers have actions which are associated with views, and you learned how to use routes and the `link_to` helper method to connect sections of the site together.

Exercises

1. Create a new page called “Contact” that contains basic contact information.

   a. Add a controller action for this page. Don't re-run the generator, though. Just edit `app/controllers/home_controller.rb` and add a new action called `contact` using the same syntax as the `about` action you see in that file.

   b. Create a view called `app/views/contact.html.erb`

   c. Add a route for this new page called `/contact`

   d. Add a link to this new page in your navigation bar.

Now, let's turn our attention to creating a simple interface so we can enter some recipes.

Chapter 3. Adding Recipes

Now that we've gotten comfortable with how controllers and views work in Rails applications, it's time to move onto working with databases. Since we're writing a cookbook, the logical place to start would be with adding and viewing recipes. And coincidentally, that happens to be the next set of features on our list:

- As a visitor, I want to be able to view recipes so I can make something good for dinner.
- As a site owner, I want to be able to create new recipes so I can share them with the world.
- As a site owner, I want to be able to modify existing recipes so I can fix my mistakes
- As a site owner, I want to be able to delete recipes so I can remove junk I don't want to share anymore.

In this chapter we'll use a feature of Rails called Scaffolding to quickly handle all of these stories. Then we'll dig deeper into how Rails works and discuss the pros and cons of this approach.

3.1. Creating the Recipes interface using Scaffolding

As you've seen already, Rails uses *generators* to help us do some common tasks. We've already used a generator to create a controller and some static pages, and now we'll use a generator to create a *scaffold*, which is a bare-bones user interface for a single database table.

Scaffolding is the much hyped but poorly understood feature of Rails. It's meant to be a starting point and a learning tool that creates a quick interface where you can do some simple testing and get some of the mundane repetitive code written for you. Scaffolding creates all the code necessary to create, update, delete, and view records in a database through a web page, including all of the HTML forms and data binding. Normally, this would take a lot of time to do correctly. With Scaffolding, we can do it in seconds.

However, scaffolding can only take you so far and is not meant for use in production, hence the name "scaffolding". For example, you can only create an interface to a single database table; multiple tables are not supported.

In this book we'll start off with scaffolding so you see how Rails works. Scaffolding is included in Rails as a learning tool. You're expected to study the code that's generated for you so you can understand how models, controllers, and views interact in a Rails application. And then, since scaffolding isn't meant for production use,

we'll clean up the code the generator makes. Then we'll move away from it as our app gets more complex.

The Scaffold Generator

Our user stories for this iteration call for an interface to manage recipes in our system, so let's use scaffolding to knock those stories off our list so we can get some user feedback.

The scaffold generator creates a model, controller, a set of views, and a migration, or a table definition. At the command prompt, ensure you're in the `cookbook` project folder.

The `generate scaffold` command takes several parameters. The first parameter is the name of the entity, or `model` you're going to create a scaffold for. Model names are singular. The generator will use this model name to create a controller, some views, and a definition for a database table. All of these, by convention, will be pluralized.

The rest of the arguments define our database table structure. We specify each field along with its data type. The scaffold generator also uses this information to build the web forms your users will see. They won't be pretty but they will work.

In your console, type the following command, all on one line:

```
$ rails generate scaffold recipe title:string
ingredients:text instructions:text
```

Watch Your Spelling!

If you misspell any of the field names here, you're going to end up with a world of problems. If you make a mistake, use the `rails destroy scaffold recipe` command and start over.

The generator runs, creating the following output:

```
invoke  active_record
create    db/migrate/20140629034739_create_recipes.rb
create    app/models/recipe.rb
invoke    test_unit
create      test/models/recipe_test.rb
create      test/fixtures/recipes.yml
invoke  resource_route
  route    resources :recipes
invoke  scaffold_controller
create    app/controllers/recipes_controller.rb
invoke    erb
create      app/views/recipes
create      app/views/recipes/index.html.erb
create      app/views/recipes/edit.html.erb
create      app/views/recipes/show.html.erb
create      app/views/recipes/new.html.erb
create      app/views/recipes/_form.html.erb
invoke    test_unit
create      test/controllers/recipes_controller_test.rb
invoke    helper
create      app/helpers/recipes_helper.rb
invoke      test_unit
create        test/helpers/recipes_helper_test.rb
invoke    jbuilder
create      app/views/recipes/index.json.jbuilder
create      app/views/recipes/show.json.jbuilder
invoke  assets
invoke    coffee
create      app/assets/javascripts/recipes.js.coffee
invoke    scss
create      app/assets/stylesheets/recipes.css.scss
invoke  scss
create    app/assets/stylesheets/scaffolds.css.scss
```

The generator created a whole bunch of components, as you can see from the output. Let's look at what we got.

The Recipe Model

The generator created a `recipe` model at `app/models/recipe.rb`. The model will interact with the database, and it'll also hold business logic. It's where we'll set up associations between tables and where we'll place code to validate user input.

A model in a Rails application is a class. This model is backed by a database table called `recipes`, and so an instance of this model will be mapped to a record in our database table.

The Recipes Controller

The generator also created a controller called `RecipesController` at `app/controllers/recipes_controller.rb`. The controller contains all of the logic that handles user requests, and interacts with the models to generate responses that the user will see. It’s got code to handle creating, editing, listing, viewing, and deleting of recipes. Because these are all common tasks, the generators can do a pretty solid job of handling this for us solely on the information we provided.

The Recipes user interface

The generator also created a folder called `app/views/recipes` which contains the HTML files for listing, displaying, creating, and editing recipe records.

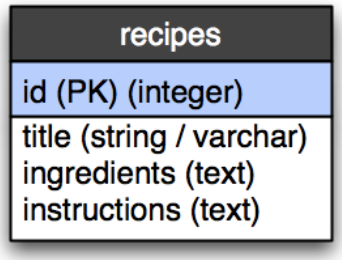
Controller and Model tests

The generator creates unit tests so that we can ensure that our models and controllers act the way we expect them to. Once we have a little more experience with how Rails works, we'll look at how we can write some of these tests. The generator gives us a start though, so we don't need to create these files ourselves.

The Recipes Table Definition

The model we just created requires a database table called “recipes”, which you can see in Figure 3.1, “The Recipes table” on page 28. Normally, you’d go and create that database table using some sort of SQL statement or visual tool. But in Rails, we use a built-in process called *migrations* to define tables and manage any updates we want to make. This lets us keep the schema of our database in sync with our code. But before we can talk about how those work, we need to talk about how Rails works with databases.

Figure 3.1. The Recipes table



The Recipe model maps to a “recipes” table and assumes the table has a primary key called ID.

> **Scaffolding Issues**
>
> Scaffolding is not dynamic. Now that we’ve generated these files, we can’t rely on scaffolding any more. Any manual changes we make would be destroyed if we attempted to run the scaffold generator again. That means that if we change the table, we’ll need to modify the views. That’s okay though because we already have a good starting point.

Databases and Rails

Open the file `config/database.yml` and review the contents of the file. It should look something like this:

Example 3.1. code/03_recipes/cookbook/config/database.yml

```
# SQLite version 3.x
#   gem install sqlite3
#
#   Ensure the SQLite 3 gem is defined in your Gemfile
#   gem 'sqlite3'
#
default: &default
  adapter: sqlite3
  pool: 5
  timeout: 5000

development:
  <<: *default
  database: db/development.sqlite3

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  <<: *default
  database: db/test.sqlite3

production:
  <<: *default
  database: db/production.sqlite3
```

This is a YAML file. (rhymes with camel) It’s a structured configuration format that maps directly to nested hashes in Ruby and is very common for configurations in Ruby on Rails. Tabs *must not be used* in YAML files. Instead, two spaces are used for each indentation.

Set Your Spaces

Now would be a good time to ensure that the text editor you're using has *soft-tabs* instead of regular tabs. That means that when you press the **Tab** key, your editor inserts spaces, not Tab characters. And, to ensure that your Ruby code fits in with that of other Ruby developers, you want to set your tabs to 2 spaces.

Let's take a look at the contents of this file:

- *Adapter* is the database adapter that we want to use. Examples are `mysql`, `sql_server`, `oracle`, `postgresql`, and `sqlite3`. We’re using `sqlite3` because it’s easy for beginners, requires no setup, and is the default database for a new Rails project. By “no setup”, I mean that you don't need to create a database, set up users, or grant permissions. Rails takes care of all of that automatically.

- *Database* is the name of the database. In this case, it’s the path to the database file. Other complex adapters would have you specify the database name or Oracle TNSNames entry here, and then you would have host, username, and password fields as well.

We have the opportunity to use lots of databases with Rails, but the default is the SQLite database. Let's look at why.

SQLite

SQLite is a simple, self-contained, serverless, zero-configuratio, transactional database. It is the most widely deployed database engine in the world thanks to its wide use in iOS and Android applications.

A SQlite database consists of a single file, much like a Microsoft Access database, but without the forms, reporting, or graphical interface. SQlite runs on all platforms and has a very standard SQL interface.

If we use SQLite with Rails, we don't need to create a database before we start our project; Rails will create the database file for us when we run our first migration. Other databases like MySQL or Microsoft SQL Server require that the database (or schema) exist and that the appropriate privileges are applied. Since we're trying to get to know Rails, we want to keep the momentum going. Using SQLite as a database makes it really simple to create a working rapid prototype. You can then move to a different database later, because you'll define your database tables using pure Ruby code instead of database-specific SQL DDL statements.

And thanks to the way that Rails works, moving from SQlite to another database won't involve having to rewrite your code.

Migrations

Migrations are used to modify your database. You use them to execute DDL statements against your database system. One of the best things about them is that they allow you to define your database as it changes; you can roll your changes back if they don't work without worrying about goofing up your database.

They're also an invaluable tool when moving to production. Migrations are supported by all of the Rails database adapters. This means that you can change database systems and apply the migration to the new database which will create your structures for you. This eliminates the need to know the various dialects of data definition languages that may change across database systems. Developers can test with SQLite3, develop with MySQL, and deploy to Oracle.

The migration file

Open the file `db/migrate/XXXXXXXX_create_recipes.rb`. The XXXXXXX part will be a numerical timestamp for the moment in time the file was created. This timestamp will help the Rails Migration system determine if it's been applied, and it also allows multiple developers to modify an application's schema without creating bottlenecks. It's contents should resemble this:

Example 3.2. code/03_recipes/cookbook/db/migrate/20140629034739_create_recipes.rb

```
class CreateRecipes < ActiveRecord::Migration
  def change
    create_table :recipes do |t|
      t.string :title
      t.text :ingredients
      t.text :instructions

      t.timestamps
    end
  end
end
```

Rails uses the information in this file to create a ‘recipes’ table in the database. Note that the above definition does not include a primary key field. Unless you specify otherwise, Rails will create an "id" column automatically, and will mark it as a primary key.

Why is the table name “recipes” and not “recipe”? Remember that by default, Rails likes table names to be the plural form of your model. It’s pretty smart too because it can do things like person => people and category => categoies. This isn’t mandatory but if we follow these conventions, we can save a few lines of code and skip a few steps. Rails will automatically look for the recipes table when we access the Recipe model in our code.

Creating the table from the migration

At this point, the table doesn't actually exist in our database - we just have the “blueprint” for it in Ruby code. To execute the migration, we'll run the command

```
$ bundle exec rake db:migrate
```

from our command line. Run that command and you’ll see feedback stating that our recipes table was created.

```
==  CreateRecipes: migrating ==================================================
-- create_table(:recipes)
   -> 0.0020s
==  CreateRecipes: migrated (0.0022s) =========================================
```

This command made a connection to the database, converted the Ruby code into a `CREATE TABLE` statement, and then executed that statement against the database. Now we've got a place to store our data.

Using The Interface

That's all we have to do in order to create, retrieve, update, and delete records in Rails. We can start the built-in server again and test out our application. At the terminal, enter

```
$ rails server
```

and navigate to http://localhost:3000/recipes in your browser.

Notice the `recipes/` in the URL. The scaffold generator created a new route for us, and the interface to work with recipes is located at this new URL. When you go there, you'll find a complete user interface for managing recipes, all created by the scaffolding, without us having to write a line of actual code.

Use this interface to create few recipes, and continue with the tutorial. The application works, but it’s a long way from good.

3.2. How did we get all that?

When we generated the Recipe scaffold, the generate created a whole bunch of Ruby code for us. It used the information we provided to create the views for our apps, including all of the data entry screens, and wrote the code in the controller that fetches the data from the models, and passes it to the views.

So instead of having to write code to connect to a database and then build data entry forms, we can use the scaffolding feature of Rails as a starting point. This is just the beginning though. There's a lot more to Rails than just scaffolding an application from a single table.. In fact, most professional Rails developers don’t use scaffolding at all because it's so limited. However, scafollding is a great way for a beginner to learn how Rails works. Let's dig deeper and examine the code the generator made so we can learn how Rails connects models, views, and controllers.

3.3. What You Learned

In this chapter you learned about scaffolding, and how it can take you from almost nothing to an entire web-based interface backed by a database. You also learned about the REST-based design pattern that Rails uses to create, retrieve, update, and delete records in a database.

Exercises

1. Create a new Rails application called "todolist" in a different folder using

   ```
   $ rails new todolist
   ```

2. Open the `Gemfile` and remove the `jbuilder` gem.

3. In the `todolist` folder, use a scaffold generator to create a database schema, model, controller, and views for an `Item` with the following fields

   a. name (string)

   b. description (text)

   c. completed (boolean)

4. Review the generated controller. You should see that it is much simpler; it doesn't have the additional code to handle JSON responses.

5. Open the newly created Migration file in the `db/migrate` folder for the `items` table. Look at the definition for the `completed` column and change it so its default value is false:

```
t.boolean :completed, default: false
```

6. Open the file `config/routes.rb` and configure it so that the `root` route points to the `index` action of the `items` controller.

7. Run the command

```
$ bundle exec rake db:migrate
```

   to create the database.

8. Start up the new app on port 4000 with the command

```
$ rails server -p 4000
```

9. Visit http://localhost:4000 to see the items page. Verify that you can create a new item.

10. Stop the server.

Now that you're comfortable using the scaffolding command and configuring a new Rails application, it's time to dig into a little more advanced stuff, and to do that, you need to learn a little more about how the Ruby programming language works. Let's dig in.

Chapter 4. Ruby Basics

Ruby on Rails is a very powerful framework, but in order to do anything even remotely advanced you should have some understanding of how the Ruby programming language works. Understanding Ruby's syntax and basic concepts will make it easier for you to follow what Rails does.

Ruby is a powerful language with a simple syntax. In this chapter, you'll become familiar with basic Ruby syntax as you work with variables, strings, arrays, and hashes. You'll learn basic control flow and how to work with objects and methods, so that when we switch back to working on our application, you'll have a better idea of where you're going.

4.1. History and Philosophy

Ruby is a dynamic, reflective, general purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features. Ruby originated in Japan during the mid-1990s and was initially developed and designed by Yukihiro "Matz" Matsumoto. It was influenced primarily by Perl, Smalltalk, Eiffel, and Lisp. The creator wanted it to be more powerful than Perl, and more object-oriented than Python.

Ruby is designed first and foremost for programmer productivity and happiness. It differs from other programming languages because of this core philosophy. Some things will be simpler, and other things will be so intuitive that you'll miss them because you'll expect it to be more difficult than it actually is.

Principle of Least Surprise

Ruby is designed to minimize confusion for experienced users. It has sensible defaults and many method aliases. For examle, you'll find things like `indexes` and `indices` as method names, and these exist so you as a developer can almost "guess" what the methods will be.

Features of the Ruby Language

- Vibrant, connected, and enthusiastic community focused on software craftsmanship and testing
- Implemented on all major platforms
- Interactive Ruby Shell (a REPL)
- Centralized package management through RubyGems
- Large standard library

- Literal notation for arrays, hashes, regular expressions and symbols
- Embedding code in strings (interpolation)
- Default arguments
- Four levels of variable scope (global, class, instance, and local) denoted by sigils and capitalization
- Automatic garbage collection
- First-class continuations
- Strict boolean coercion rules (everything is true except false and nil)
- Exception handling
- Operator overloading
- Built-in support for rational numbers, complex numbers and arbitrary-precision arithmetic
- Custom dispatch behavior (through method_missing and const_missing)
- Native threads and cooperative fibers
- Full support for Unicode and multiple character encodings (as of version 1.9)
- Native plug-in API in C
- Thoroughly object-oriented with inheritance, mixins, and metaclasses
- Dynamic typing and Duck typing
- Everything is an expression (even statements) and everything is executed imperatively (even declarations)
- Succinct and flexible syntax that minimizes syntactic noise and serves as a foundation for domain specific languages
- Dynamic reflection and alteration of objects to facilitate metaprogramming
- Lexical closures, Iterators and generators, with a unique block syntax

4.2. Interactive Ruby

You can work with many of the code examples in this part of the workshop by using `irb`, or Interactive Ruby. From your command prompt or terminal window, type `irb` and you should then see this:

```
irb(main):001:0>
```

That's the IRB prompt, and when you see it, you can type in a Ruby expression and hit the Enter key to see the results. Try out a few Ruby expressions here, and you'll immediately see the results:

```
$ irb
irb(main):001:0> 5 + 2
=> 7
irb(main):002:0> 5 * 2
=> 10
irb(main):003:0> Date.today.year
=> 2010
```

You should try to work through the examples in this chapter in `irb` to get comfortable with the language. Don't just copy and paste the code. Type it out to get a feel for the Ruby language.

4.3. Numbers, Strings, Variables

Let's start exploring Ruby by working with basic data types, starting with numbers.

Numbers

Numbers are simply declared either as whole numbers or decimals.

Example 4.1. code/ruby/data_types.rb

```
32                          # Fixnum
151239123491234192349123 4 # Bignum
32.50                       # Float
3.14159                     # Float
```

As with any programming language, we can do simple math with these numbers.

Example 4.2. code/ruby/data_types.rb

```
1 + 2 # addition
5 - 4 # subtraction
2 * 2 # multiplication
6 / 3 # division
15 / 2 # division without remainder
15 % 2 # modulo (remainder)
```

Addition, multiplcation, division, subtraction work exactly as you'd expect them to work.

Strings

Groups of characters are called Strings, and you see them with single and double quotes in Ruby.

Example 4.3. code/ruby/data_types.rb

```
"Homer"
'Homer'
%Q{I don't need to worry about "quotes" in my string.}
```

We can also add, or concatenate strings together. In Ruby, we concatenate strings using the plus sign, like this:

Example 4.4. code/ruby/data_types.rb

```
"Homer" + " Simpson"
```

although we often use alternative methods for string concatenation, as you'll see later.

4.4. Variables

Variables in Ruby work like variables in other languages, except that we don't define the data type explicitly. Ruby is not statically typed, but it *is* strongly typed. This means a variable's data type is determined by the value it's assigned.

Example 4.5. code/ruby/data_types.rb

```
name = "Homer"
age = 49
```

Because of this, it is possible to reuse a variable for a different data type. If you're used to a static language like Java, this might make you uncomfortable. However, in Ruby, we're not terribly concerned about types. Because everything is an object, a Ruby developer is more concerned about what the object can do than what the object is.

Variable Names

We have to follow some naming conventions when we create variables. These variable names are good. Try them out in your IRB session.

Example 4.6. code/ruby/data_types.rb

```
name = "Homer"
_name = "Homer"
first_name = "Homer"
first__name = "Homer"
```

There are some characters you can't use.

Example 4.7. code/ruby/data_types.rb

```
1stname = "Homer" #starts with a number
first name = "Homer" # invalid because it has a space
first-name = "Homer" # invalid because it looks like subtraction
```

These variables we've created are all local variables. They can only be seen in the current method. Other parts of our program might not be able to see them. When you're working in IRB, you're actually running inside of a method, and the variables you create are all local to that method. Don't worry too much about that just yet. We'll talk about methods and scope very soon.

Constants

You can define variables that start with a capital letter, but Ruby treats those as Constants. Remember that a Constant is a variable that, once set, cannot be changed later.

Example 4.8. code/ruby/data_types.rb

```
puts ENV["HOME"]            # display your home folder
WINDOWS_DIR = "c:/windows"  # set a constant
puts WINDOWS_DIR            # get the value of a constant
```

Ruby uses constants to refer to classes as well. The `Date` class is just one example of that.

Example 4.9. code/ruby/data_types.rb

```
require 'date'
puts Date.today             # Call the 'today' class method on the
                            #   Date class
```

When we define our own classes later, we'll define those classes as constants as well.

Data Types

As Ruby is dynamically typed, variables don't have types, but the objects that we assign to those variables do. For example, here are three different ways we define arrays in Ruby. Each one of these createes an Array object.

Example 4.10. code/ruby/data_types.rb

```
names = ["Homer", "Marge", "Bart"]
colors = Array.new
cities = []
```

Ruby is strongly typed, which means that if you want to add a number to a string, you have to turn the number into a string using the `to_s` method. The data you're adding has to be of the same type.

Example 4.11. code/ruby/data_types.rb

```
name = "Homer"
number = 2
puts name + " has " + number.to_s + " donuts"
```

The most common place you run into issues with this is when you're building up strings for display. Thankfully there's a great shortcut.

String Interpolation

When we use double-quoted strings, we can use the #{} notation within the double quotes to embed expressions within the string. Since every expression in Ru-

by returns a value, we can easily embed the return value of any expression within a string.

Example 4.12. code/ruby/data_types.rb

```
name = "Homer"
number = 2
puts "#{name} has #{number} donuts"
```

This is the preferred way to create strings that use variables in Ruby, and you'll use it a lot.

Automatic String Conversion

Expressions embedded in strings with #{} are automatically converted to strings, so you won't have any type conversion errors when you use this method.

4.5. Logic

Programs are boring without logic. Here's how we do logic in Ruby.'

Comparison operators

Like Java, C#, and JavaScript, Ruby uses = for assignment and == for comparison.

Example 4.13. code/ruby/data_types.rb

```
5 == (2+3) # equality
5 != (2*3) #inequality
5 < 10 # less than
5 <= 5 #less than or equal to
5 >= 5 # greater than
10 > 5 # greater than or equal to
```

Control Flow

We control how programs work by evaluating expressions and making decisions. Ruby has several control flow statements we can use.

If

Like most languages, Ruby has the if statement to control program flow.

Example 4.14. code/ruby/logic.rb

```
if age < 16
  puts "You are not old enough to drive."
end
```

But Ruby can do something a little interesting - it understands `if` statements at the *end* of the line too.

Example 4.15. code/ruby/logic.rb

```
puts "You are not old enough to drive." if age < 16
```

This is occasionally very handy and something you'll see in others' code.

If..Else

Sometimes you need to make a choice in your program, and you do that like this:

Example 4.16. code/ruby/logic.rb

```
if age < 16
  puts "You are not old enough to drive."
else
  puts "You are old enough to drive."
end
```

Unless

Ruby also introduces `unless` to improve readability.

Example 4.17. code/ruby/logic.rb

```
unless age < 16
  puts "You are old enough to drive."
end
```

This can also be used as a predicate.

Example 4.18. code/ruby/logic.rb

```
puts "You are not old enough to drive." unless age >= 16
```

4.6. Methods, Classes, and Objects

When programs get more complex, we organize things into methods. We can then organize those methods inside of classes that represent things.

Methods

A method, sometimes called a function, organizes a bit of code that is reusable. Good methods only perform one task. Here's a method that creates a simple greeting based on the given first and last names:

Example 4.19. code/ruby/methods.rb

```
def hello(first_name, last_name)
  "Hello #{first_name} #{last_name}!"
end

# calling the method and printing to the screen
puts hello("Homer", "Simpson")
```

Unlike other languages, Ruby doesn't require us to use the `return` keyword to return a value from a function. The last statement in a function becomes the return value.

Here's a slightly more complicated method. This method computes the number of gallons of paint needed to paint a rectangular ceiling. It assumes that one gallon of paint covers 350 square feet:

Example 4.20. code/ruby/methods.rb

```
def gallons_needed_for_ceiling(length, width)
  area = length.to_f * width.to_f  #convert to floats
  area / 350
  area.ceil      # round up and return value
end

puts gallons_needed_for_ceiling(40, 40)
```

This method converts the input values to floating point numbers using the `to_f` method. It also uses the `ceil` method to round the number up to the next whole number.

Methods are always defined as part of an object. Even the methods we just defined are part of an object called `Object`, which every object is based on. And to define our own objects, we use Classes.

Classes

Objects represent real entities in our systems, and they contain methods. We use Classes to define these objects, just like we would in Java or C#. Here's how we define a class in Ruby that lets us manage the properties of a Person.

Example 4.21. code/ruby/classes.rb

```
class Person
  @last_name
  @first_name

  # getter methods
  def first_name
    @first_name
  end

  def last_name
    @last_name
  end

  # setter methods
  def first_name=(first_name)
    @first_name = first_name
  end

  def last_name=(last_name)
    @last_name = last_name
  end
end
```

We define our object, and then we create instance variables and create getters and setters for those instance variables. In Ruby we call these “accessor methods”.

Here's how we use this new class:

Example 4.22. code/ruby/classes.rb

```
homer = Person.new
homer.first_name = "Homer"
homer.last_name = "Simpson"
homer.first_name
homer.last_name
```

First we create a new instance of the object. Then we call the accessor methods to set or retrieve the values.

Notice that we're using the equals sign to set the values of these properties; it's part of the method name! There is no distinction made between methods, properties, and fields (or "member variables") in Ruby, and this method naming convention makes it look more like you're setting properties. Rubyists don't like the “get” and “set” prefixes used in Java and other languages. Instead, Rubyists use this parallel naming scheme.

Now, in this above example we've created two instance variables and four accessor methods to let us set and retrieve the values. This is a lot of code, and it's such a common pattern that Ruby extracts it away from us. We can rewrite our class so it looks like this instead:

Example 4.23. code/ruby/classes.rb

```
class Person
  # getter and setter methods defined as
  # accessors!
  attr_accessor :first_name, :last_name
end
```

This works for most simple cases. The `attr_accessor` method creates getter and setter methods for us, as well as the instance variables. Of course, if you needed to be more specific, you can always write certain methods out by hand yourself. It's good to have options!

Ruby also gives us `attr_reader` which only creates the "getter" method, and `attr_writer` which only creates the "setter" method. When you're building your objects, you can use this approach to create your classes, or you can type out the methods yourself when you need them to do a little more than just setting or retrieving values of attributes.

Instance Variables

Instance variables start with the @ character. Unlike local variables, all instance methods in a class have access to these variables. However, they are considered “private” by default; nothing outside the instance of the object can access those variables without going through a method.

Instance Methods and Class Methods

The types of methods you've defined are instance methods. That means you have to create an instance of the class to call the methods.

Sometimes we need to have methods we can call that do not require an instance. In some languages these are known as static methods. In Ruby we call these class methods. We define these methods like this:

Example 4.24. code/ruby/classes.rb

```
class Pdf
  def self.write
    # some code that writes a PDF
  end
end

Pdf.write
```

In Ruby, everything is an object, even classes themselves. Thus, classes can have methods.

You'll use lots of class methods in Ruby, and in Rails. Previously, when you created a new instance of your `Person` object, you used `Person.new` , which is a class method which the `Person` object gets from its parent.

Inheritance

Ruby supports inheritance just like Java. An object may inherit from a single parent class. In Rails, you'll find quite a bit of this. For example, models that represent database tables inherit from a parent class called `ActiveRecord::Base`:

```
class Recipe < ActiveRecord::Base
end
```

and Controllers often inherit from `ApplicationController` which inherits from `ActionController::Base`

In case you're wondering, the double colon you see in `ActionController::Base` and `ActiveRecord::Base` is what's called a Module, which is a way that Ruby lets us group methods and classes. It's similar to a Namespace or a Package in other languages.

The `<` symbol denotes inheritance in Ruby. It's the equivalent of the `extends` keyword in other languages.

As you explore Ruby on your own, you'll learn more about modules and how you can use them to group behavior. You'll also learn how modules can help you solve problems that are usually solved by inheritance in a cleaner, more streamlined fashion. But that's out of scope for this book.

4.7. Arrays and Hashes

Ruby arrays are collections of objects.

Example 4.25. code/ruby/arrays_and_hashes.rb

```
colors = ["Red", "Green", "Blue"]
```

Hashes are also collections, but they let us associate the objects with keys.

Example 4.26. code/ruby/arrays_and_hashes.rb

```
attributes = {:age => 25,
              :first_name => "Homer",
              :last_name => "Simpson"}
```

You often see hashes used in Rails as method parameters because they make things more self-documenting.

Example 4.27. code/ruby/arrays_and_hashes.rb

```
class Box
  attr_accessor :height, :width, :color
  def initialize(options)
    @height = options[:height]
    @width = options[:width]
    @color = options[:color]
  end

  def to_s
    "A #{color} box, #{width} cm wide by #{height} cm high."
  end
end

puts Box.new(:height => 2, :width => 2, :color => "Green")
puts Box.new(:color => "blue", :width => 2, :height => 4)
```

The nice thing about this is that the order doesn't matter and you don't need an API to interpret the method call.

Symbols

Symbols are special objects that we use to label things like hash keys. They're like constants, but they can't have a value. We use them to conserve memory.

Since everything in Ruby is an object, each string we create takes up memory. These lines of code create 18 objects total.

Example 4.28. code/ruby/arrays_and_hashes.rb

```
homer = {"first_name" => "Homer", "last_name" => "Simpson"}
bart = {"first_name" => "Bart", "last_name" => "Simpson"}
marge = {"first_name" => "Marge", "last_name" => "Simpson"}
```

We can use symbols for the hash keys and reduce the object count by 4.

Example 4.29. code/ruby/arrays_and_hashes.rb

```
homer = {:first_name => "Homer", :last_name => "Simpson"}
bart = {:first_name => "Bart", :last_name => "Simpson"}
marge = {:first_name => "Marge", :last_name => "Simpson"}
```

It doesn't seem like a lot, but what if we had a huge codebase that used those arrays over and over?

4.8. Blocks

Blocks let us evaluate code in the context of other code. They are anonymous functions, or closures, and they're used everywhere in Ruby.

Here's the simplest thing you can do with a Block. Integers have a method called `times` that will repeat code. To print out the words “Hello World” five times, we can use an integer `5`, the `times` method, and a block, like this:

Example 4.30. code/ruby/blocks.rb

```
5.times do
  puts "Hello"
end
```

We also often use blocks to iterate over entries in arrays like this:

Example 4.31. code/ruby/blocks.rb

```
colors = ["red", "green", "blue"]

colors.each do |color|
  puts color
end
```

This often removes the need for a standard For loop that you'd use in other languages. But blocks are more powerful than that. We can use them to check if our array contains something that meets a certain condition:

Example 4.32. code/ruby/blocks.rb

```
colors.find do |color|
  color == "red"
end
```

Here we are just checking to see if the array contains the color "Red". This statement returns `"red"` if our array of colors contains the word “Red” and `nil` if it does not.

Blocks have an alternative syntax that developers often use if the code within the block is only one line. We simply change the first `do` to an opening curly brace, and the `end` to a closing curly brace. The result looks like this:

Example 4.33. code/ruby/blocks.rb

```
colors.find{|color| color == "red"}
```

And we can even use blocks to create new arrays from existing arrays. Here's an example where we can loop over an array and extract only the even numbers into a new array:

Example 4.34. code/ruby/blocks.rb

```
numbers = [1,2,3,4,5,6,7,8]
numbers.select{|number| number % 2 == 0}
```

But we can also use them to execute code later, so we can create nice looking DSL. Here's how we create database tables in Rails:

Example 4.35. /Users/brianhogan/books/cookbook2/code/ruby/blocks.rb

```
create_table :products do |t|
  t.string :name
  t.text :description
  t.boolean :visible
  t.integer :price
  t.timestamps
end
```

You will constantly use Blocks in Ruby.

4.9. Rules of Ruby

When developing with Ruby, you can avoid a lot of problems if you remember these rules:

Everything is an object

Everything in Ruby is an object, even things like Integers and Strings which are primitives in other languages. That means they all have methods.

Example 4.36. code/ruby/objects.rb

```
length_of_string = "Bart".length
length_of_array = ["Bart", "Homer", "Marge"].length

5.times do
  puts "Hello world"
end
```

Even integers are objects, which mean they have methods which we can use for iteration instead of writing "for" loops.

Every statement returns a value

Every single statement in Ruby returns a value, which is why we can do things like this to shorten our code:

Example 4.37. code/ruby/logic.rb

```
message = if age < 16
            "You are not old enough to drive."
          else
            "You are old enough to drive."
          end

puts message
```

A method's implicit return value is the result of the last executed statement.

You don't need to explicitly return a value in Ruby.

Example 4.38. code/ruby/methods.rb

```
def can_drive?(age)
  age >= 16
end
```

While you can use the `return` keyword, almost nobody does.

Everything evaluates to True except Nil and False

Possibly the most important rule in Ruby. Everything evaluates to True, even 0 and -1. In fact, True, False, and Nil are actually objects themselves, with methods.

Example 4.39. code/ruby/logic.rb

```
result = 0
if result
  puts "Hello World"
end

result = false
if result
  puts "Hello World"
end
```

Be careful with this. It's easy to accidentally return a value you don't expect. Remember that the return value of a statement is the last executed statement in the method.

False and Nil are not equivalent

Keep in mind that while they won't evaluate to True, they are not equivalent.

Example 4.40. code/ruby/logic.rb

```
is_a_ninja = false
is_a_ninja.nil?       # false
```

It's in your best interests to write your own code so that it explicitly returns `true` or `false`, rather than `true` or something else.

4.10. What You Learned

Our brief tour of Ruby ends here, but we're just getting started. As you work more on your first Rails application, you'll make use of the things you explored here, including:

- Local variables
- Arrays
- Hashes and symbols
- Methods
- Classes
- Instance methods
- Blocks

In fact, you've already seen examples of most of these when we explored the code that the scaffold generator created for us.

Exercises

This is a simple program in Ruby that asks you for two numbers and adds them together, printing out the result:

Example 4.41. code/ruby/add_two_numbers.rb

```
# prints message to the screen with a line break
puts "This program adds two numbers."

# prints message without a line break
print "Enter the first number and press Enter: "

# waits for input, storing input into the variable on the left side.
first_number = gets

print "Enter the second number and press Enter: "

second_number = gets

# When the user enters a number, they'll press the enter key. It'll be saved
# as the character \n. This next line removes the last character from a
# string, effectively removing the \n character.
first_number.chop!
second_number.chop!

# convert the entered values to integers.

first_number = first_number.to_i
second_number = second_number.to_i

# Do the math and print out the result.

sum = first_number + second_number

message =  "The sum of #{first_number} + #{second_number} is #{sum}"

puts message
```

This program introduces you to the `print` and `gets` methods which deal with console input and output. Type out this program and save it as `add_two_numbers.rb`. Run this program like this:

```
$ ruby add_two_numbers.rb
```

Now use what you learned to create the following programs:

1. Create a program that prompts the user for a password. The program should compare the password given by the user to a known password. If the password matches then the program should display "Welcome". If they don't match, the program should display "I don't know you."

2. Create a program that asks for three numbers and computes a total. Use a loop to prompt for these three numbers and total up the results. Use an array to store the user's input so you can display the numbers the user entered.

3. Using the following code as a starting point, create a program that iterates over the collection of users and displays the email address for each person. The email address is their username, followed by `@veridiandynamics.com`. Display the email addresses separated by semicolons. There should be no semicolon after the last one.

   Here's the sample code:

   Example 4.42. code/ruby/people.rb

   ```
   class Person
     attr_accessor :name, :username
     def initialize(name, username)
       self.name = name
       self.username = username
     end
   end

   people = [
     Person.new("Veronica Palmer", "vpalmer"),
     Person.new("Ted Crisp", "tcrisp"),
     Person.new("Linda Zwordling", "lzwordling"),
     Person.new("Phil Myman", "pmyman")
   ]
   ```

   Use the `each` method to iterate over the array, or investigate the `collect` or `map` method in the Ruby documentation to learn how to create a new array from the existing array. Investigate the `join` method which can convert an array of items into a string, separated by the character you pass in.

Now that you have some basic Ruby under your belt, let's get back to working with Rails. We've used scaffolding to build a section of our app, and even though it works, that scaffold generator made a bit of a mess. Let's dig in to how the scaffolding works, and see if we can clean it up.

Chapter 5. Exploring and Cleaning Up the Scaffolding

The scaffolding needs some work before it can be used in production. As you become more familiar with the way Rails works, you will rely less and less on scaffolding to create your controllers and views because you'll want flexibility that scaffolding simply doesn't provide. But before you can go off on your own, you'll need to understand how Rails applications work, and exploring the scaffolding is a great and fast way to do that. Armed with our knowledge of Ruby, let's explore how the scaffolding worked and then make some improvements.

5.1. Scaffolding and The Rails Way

With a single command, the scaffold generator gave us a model, a controller, a database definition, and a user interface. By leveraging some Ruby, a handful of conventions, and the information we provided, Rails built everything we need to make a simple working application. But this isn't anything we couldn't code by hand, and before we get any farther, we need to look at how everything the scaffolding gave us fits together, because this is the pattern that most Rails applications use when it comes to managing records in a database, and you must understand it before you can move on to more advanced things.

Listing Recipes

Let's start by looking at how the generated code builds the list of recipes that we see when we visit http://localhost:3000/recipes. When we visit that URL, the Rails router takes us to the Recipes controller's `index` action, thanks to a route that the Scaffold generator added for us.

Open up `app/controllers/recipes_controller.rb` and look at the `index` action.

Example 5.1. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes
# GET /recipes.json
def index
  @recipes = Recipe.all
end
```

A controller is a special Ruby class that inherits from a Rails controller parent class. Actions are simply the public methods in a Controller class.

In the `index` method of this `RecipeController` class, we grab all of the recipes from the database, using our model, and put them in an instance variable.

Then, the view `app/views/recipes/index.html.erb` gets rendered automatically. We don't have to explicitly tell Rails we want to render that view. This is another example of convention over configuration. Rails assumes that we want to render a view with the same name as the controller's action. In this case we're in

the Recipes controller, in the `index` action, so Rails looks in the `views/recipes` folder for a file called `index.html.erb`. The code in that view displays a table of the recipes.

Example 5.2. code/03_recipes/cookbook/app/views/recipes/index.html.erb

```
<h1>Listing recipes</h1>

<table>
  <thead>
    <tr>
      <th>Title</th>
      <th>Ingredients</th>
      <th>Instructions</th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>

  <tbody>
    <% @recipes.each do |recipe| %>
      <tr>
        <td><%= recipe.title %></td>
        <td><%= recipe.ingredients %></td>
        <td><%= recipe.instructions %></td>
        <td><%= link_to 'Show', recipe %></td>
        <td><%= link_to 'Edit', edit_recipe_path(recipe) %></td>
        <td><%= link_to 'Destroy', recipe, method: :delete, data: { confirm: 'Are
 you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Recipe', new_recipe_path %>
```

However, if the end user requested JSON, then Rails would serve a JSON file. If you visit http://localhost:3000/recipes.json, you'll be able to see your recipes automatically rendered as JSON which you could consume in another service. You'll see how that's created by looking at the file `app/views/recipes/index.json.jbuilder`.

Example 5.3. code/03_recipes/cookbook/app/views/recipes/index.json.jbuilder

```
json.array!(@recipes) do |recipe|
  json.extract! recipe, :title, :ingredients, :instructions
  json.url recipe_url(recipe, format: :json)
end
```

So where does that data come from? The model.

How Models Work

Models are responsible for fetching data from the database and for getting that data back into the database, and for other business logic related to that data.

The scaffold generator also created a Ruby class called `Recipe`. But if you look at the code in the model, you'll see nothing more than the class definition!

Example 5.4. code/03_recipes/cookbook/app/models/recipe.rb

```
class Recipe < ActiveRecord::Base
end
```

What's going on here? Where's all the code that creates, retrieves, updates, and deletes records??

A Rails model is a regular old Ruby class. There's absolutely nothing special about models in Rails. However, the scaffolding creates a model that inherits from a parent class called `ActiveRecord::Base`. That parent class does all of the work. >ActiveRecord is an ORM, or an Object-Relational Mapper, which maps database records to objects.

Models Are Just Ruby

Although most models you'll encounter in a Rails application will use ActiveRecord, you must remember that a model is merely a Ruby class. A model does not need to be backed by a database or use ActiveRecord. A model is *where your business logic goes*.

Normally, Ruby classes use methods that developers define for accessing instance variables in a class. But ActiveRecord attempts to build methods dynamically based on the information in your database. The class is mapped to a database table and each field in that table becomes an accessor, which is a a Ruby term meaning that it can be read and written to. Accessors are like “getters” and “setters” in Java.

This is all done by Rails' naming conventions. Rails expects that your database table names are pluralized. Rails can then use the model's name and determine the table it should use. In our case, it takes the word “Recipe” and figures out that there's a database table called “recipes”. It literally lowercases the word “recipe” and then pluralizes it. It then looks at the schema for that “recipes” table and generates the necessary accessor methods on the fly. By using the same kind of conventions, it can also generate all of the SQL statements needed to select, insert, update, and delete records from the database.

For example, in our system, a `Recipe` has a title. An instance of our `Recipe` model will have a method called `title` which can be used as a *getter* and a *setter*. We don't have to write the code for that because Rails generates it for us. This process, called reflection, is one of the most powerful features of the Ruby programming language.

Therefore, we don't see any of these accessor methods when we look at the code for our model. We also don't see the code that Rails uses to fetch or save records. The code for that is handled by the parent class. When we say “get me all of the Recipes”, the parent class has a method called `all` that looks at the model name, figures out the table name, and uses that to generate the appropriate SQL statement. The same thing happens when we save, update, or delete a record.

This concept great for developer productivity because we're not defining fields in our database and then manually mapping them to accessors in our models, but it's bad for performance, and so in production mode, this automatic generation is only done once when the application starts. Any changes to the database or the code won't take effect until the application is restarted by a server admin.

Now let's talk about how we get the “new” , “edit” , and “delete” links on that page to work.

Understanding Resource Routing

You've already seen how Rails makes it easy to create hyperlinks using the `link_to` helper. Each view in our application is generally associated with a controller action and each URL maps to a controller and a action. When we ran the scaffold command, it created several controller actions and views which we need to link together.

Rails is all about conventions. When we generated the scaffold, the generator modified the `routes.rb` and defined routes for a `resource`. This definition creates some helper methods that make making links incredibly easy, but you first have to understand the conventions it uses.

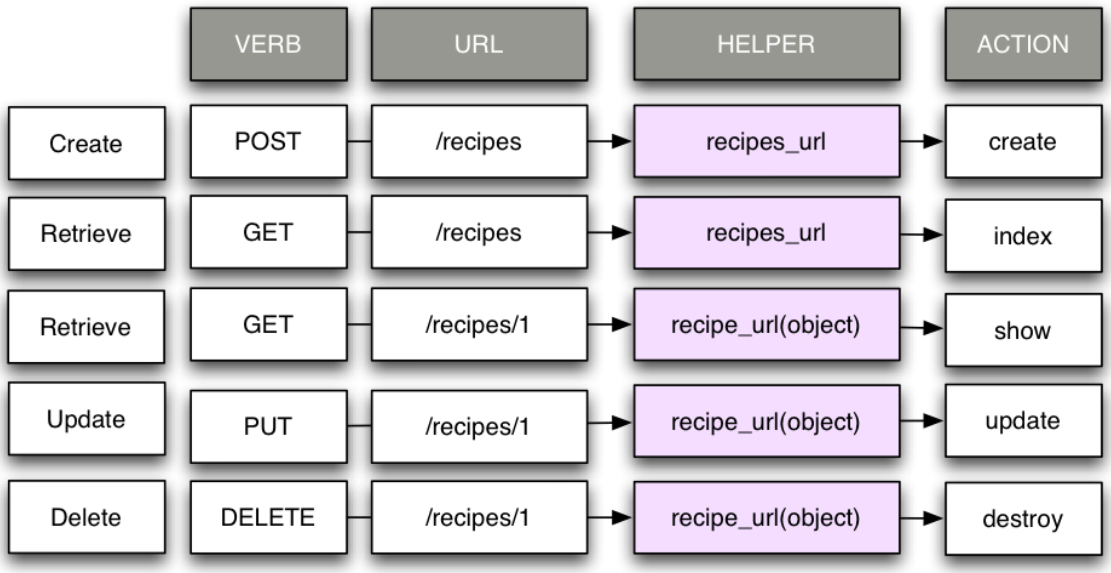
Open `routes.rb` and you'll see this near the top of the file:

Example 5.5. code/03_recipes/cookbook/config/routes.rb

```
resources :recipes
```

This one line sets up several named routes, similar to the one you created for the “About” page. The table in Figure 5.1, “Rails RESTful Routing” on page 54 shows how this resource routing works.

Figure 5.1. Rails RESTful Routing



This "Resource routing" looks at an incoming request, uses the HTTP verb (GET, POST, PUT, or DELETE), combined with the URL you specify, to identify the controller and action that needs to be called. This is a slight shift away from the default method, where every URL maps directly to a controller and an action. You could still do things that way, but this convention does speed up your work a bit. Here are some examples:

We create a link to the list of recipes like this:

```
link_to "Recipes", recipes_url
```

If we have a recipe object, we can use that recipe object to build a link.

```
link_to recipe.title, recipe_url(recipe)
```

or

```
link_to recipe.title, recipe
```

Remember earlier when we talked about CRUD? Rails is mapping CRUD to controllers using HTTP Verbs. Creates use `POST`, Retrieves use `GET`, updates use `PUT`, and destroys use `DELETE`

Hyperlinks on the web make `GET` requests. If you want to make a link to delete a record, you need to pass along the HTTP verb, like this:

```
link_to "Delete", recipe_url(recipe), :method => :delete
```

Web browsers currently can only make `GET` and `POST` requests, so Rails uses a hidden form field to specify the correct HTTP verb for you. More sophisticated libraries can and do take advantage of this mapping. This lays the groundwork for the creation of REST APIs that can be used as backends for iPhone applications, Flash frontends, and more.

New and Edit

The `new` and `edit` pages are special. This type of routing was designed to make it easier to build web services that could be consumed by other applications. The `new` and `edit` pages just display web forms for users to interact with, and so those aren't really needed by webservic es - a web service client would just call the `create` and `update` actions directly, since the form would be on the iPhone or in the Flash application. So, to make links to these pages, you'd do something like this:

```
link_to "New Recipe", new_recipe_url
```

```
link_to "Edit Recipe", edit_recipe_url(recipe)
```

If your head is spinning, don't worry too much about it yet. Use Figure 5.1, “Rails RESTful Routing” on page 54 as a reference for now, and as you work more with Rails, you'll get used to how it works

REST isn't Mandatory

You shouldn't feel like you have to use REST-style routing like this in your applications. If you find that making your own named routes, that's perfectly fine. In fact not every situation you'll develop will require all of these resource urls. Just be aware that most people who write Rails applications make heavy use of this feature when it's a good fit.

Creating A Recipe

The recipe creation process is a bit more complex than listing recipes. It involves two controller actions and two views.

New

The `new` action and view display the form that the user fills in to create a new recipe.

Example 5.6. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes/new
def new
  @recipe = Recipe.new
end
```

This creates a new instance variable which gets passed to the view.

Example 5.7. code/03_recipes/cookbook/app/views/recipes/new.html.erb

```
<h1>New recipe</h1>

<%= render 'form' %>

<%= link_to 'Back', recipes_path %>
```

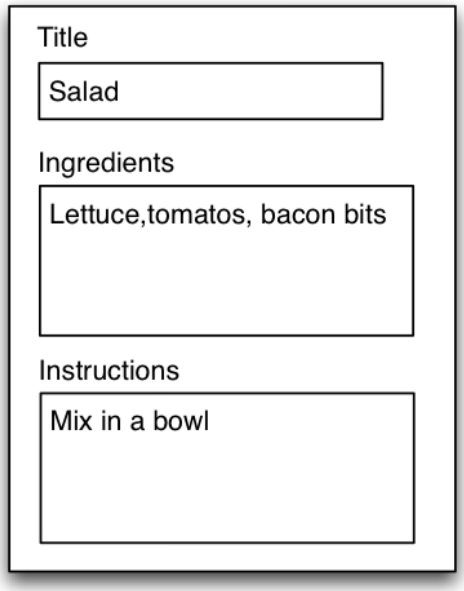
The actual web form our users will fill out isn't actually on this page. It's included by using something called a partial.

Partials

The *New* and *Edit* forms are identical except for their headings. The actual form content is shared across both files using a partial which is similar to an include file in PHP. If we have to add a field to the form, we only need to add it to one file instead of two.

In our case, our form is stored in `app/views/recipes/_form.html.erb` and looks like the example in Figure 5.2, “Our form fields” on page 57.

Figure 5.2. Our form fields




```
<%= form_for(@recipe) do |f| %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :ingredients %><br />
    <%= f.text_area :ingredients %>
  </div>
  <div class="field">
    <%= f.label :instructions %><br />
    <%= f.text_area :instructions %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

The Create flow

When the user submits the form, one of two things can happen: the save can be successful, or it can fail. The logic in our controller's `create` action handles both of those cases.

Example 5.8. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
# POST /recipes
# POST /recipes.json
def create
  @recipe = Recipe.new(recipe_params)
  respond_to do |format|
    if @recipe.save
      format.html { redirect_to @recipe, notice: 'Recipe was successfully created.'
 }
      format.json { render action: 'show', status: :created, location: @recipe }
    else
      format.html { render action: 'new' }
      format.json { render json: @recipe.errors, status: :unprocessable_entity }
    end
  end
end
```

The create action creates a new Recipe object from the data the user entered on the form, which Rails makes available in the params hash. We access this data using the `params`. Here's what the entire hash looks like when we submit the form to create a new recipe:

```
{"utf8"=>"âœ“",
          "authenticity_token"=>"SOMETHING",
          "recipe"=>{
            "title"=>"Salad",
            "ingredients"=>"Lettuce, tomatos, bacon bits",
            "instructions"=>"Mix in a bowl"},
          "commit"=>"Create Recipe",
          "action"=>"create",
          "controller"=>"recipes"}
```

The Authenticity Token

The authenticity token is generated each time the user displays the form. When the form is submitted, Rails checks this token and uses it to determine if the form request came from the actual application. This prevents other web sites from posting form data to our application. This is an example of Rails' built-in Cross Site Request Forgery protection. It protects your users, and is turned on by default. You can safely disregard this token for now.

Figure 5.3. How form fields map to the Parameters hash

The Rails form helpers we used bundle the data in a subhash of the `params` hash, with a key that matches the object. You can see how this works in Figure 5.3, “How form fields map to the Parameters hash” on page 58. We can take all of the data the user filled in and create a new object instance with one line of code:

Example 5.9. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
@recipe = Recipe.new(recipe_params)
```

When the recipe object saves successfully, we set a message in the Flash and then redirect the user to the Show page. If the save fails, we redisplay the recipe form so that the user can fix any mistakes.

Showing, Editing, and Updating records

When we're viewing and modifying an existing record, we're working with only one record at a time. When we are showing a record, we grab the record from the database by parsing the record's ID out of the URL, which we find in the `params` hash as `params[:id]`.

The `show` action retrieves the recipe from the database. But when we look at the controller, the method is blank!

Example 5.10. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes/1
# GET /recipes/1.json
def show
end
```

But if you look at the top of the controller, you'll find this line:

Example 5.11. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
before_action :set_recipe, only: [:show, :edit, :update, :destroy]
```

This is an example of a `before_filter`. We can specify that we want to fire some code *before* our controller action runs. This particular filter says that we're going to call a method called `set_recipe` whenever the `show`, `edit`, `update`, or `destroy` methods are invoked. And at the bottom of the `RecipeController` we find that `set_recipe`:

Example 5.12. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
private
  # Use callbacks to share common setup or constraints between actions.
  def set_recipe
    @recipe = Recipe.find(params[:id])
  end
```

This method looks at the params hash and grabs the id that was set there via the url. It uses that ID to call the `find` method on the `Recipe` model, which sends a request to the database. It actually generates the sql statement `select * from recipes where id = 1`. The `find` method is a *class* method of the `Recipe` class and returns an instance of a recipe object. The controller then displays it in the requested format, using the same method used by the `index` action.

We do the `edit` action in a similar fashion.

Example 5.13. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes/1/edit
def edit
end
```

When we submit the Edit form, the response is sent to the `update` action, where we fetch the record we want to update We then update its attributes from the form data we collected.

Example 5.14. code/03_recipes/cookbook/app/controllers/recipes_controller.rb

```
# PATCH/PUT /recipes/1
# PATCH/PUT /recipes/1.json
def update
  respond_to do |format|
    if @recipe.update(recipe_params)
      format.html { redirect_to @recipe, notice: 'Recipe was successfully updated.'
 }
      format.json { head :no_content }
    else
      format.html { render action: 'edit' }
      format.json { render json: @recipe.errors, status: :unprocessable_entity }
    end
  end
end
```

If it saves, we display that message to the user, and if it doesn't, we show the form again.

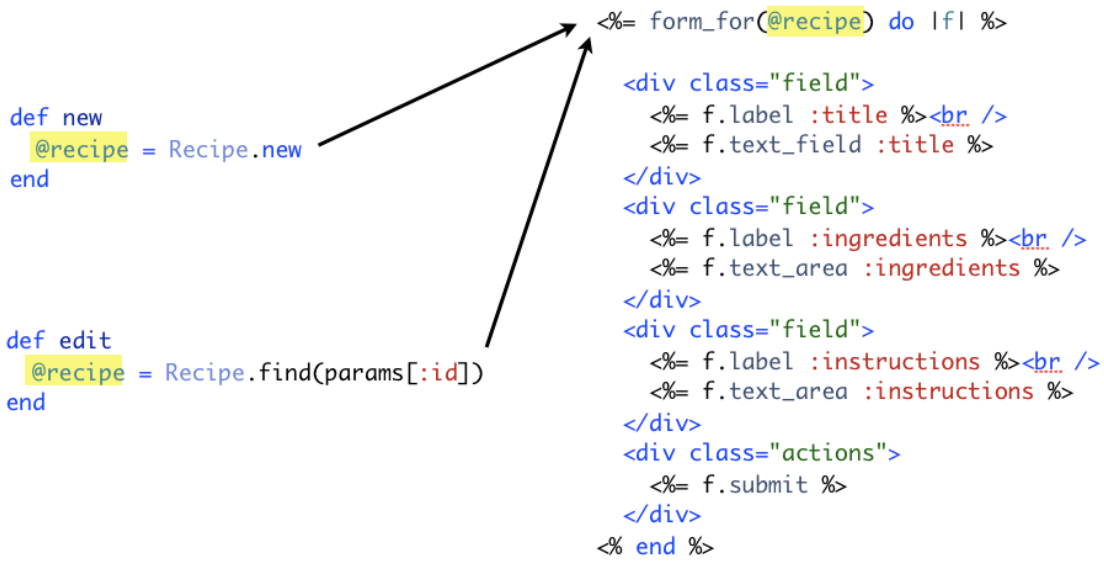
How Form Helpers Work

One thing still might not be clear. If the `new` and `edit` actions are both sharing the same form code, then how does Rails know to create a new record or update an existing one?

It's actually all just a matter of conventions. In the `new` action, we create a new empty instance of a `Recipe` object, but in the `edit` action, we fetch an existing one from the database. The `form_tag` method inspects the `@recipe` object instantiated in the `new` or `edit` action of the controller to determine whether it's a new object or an existing one, and can construct the appropriate form action. If `@recipe` doesn't have an `id` assigned, Rails assumes that it's not yet in the database and so it constructs a form that posts to the `create` action. When the `@recipe` does have an `id`, the helper constructs a form that calls the `update` action instead. Take a look at Figure 5.4, “Form helpers use objects to determine the URL and method” on page 61 for an illustration of this.

Keep in mind that none of this is really magic. It all comes down to the routing conventions we discussed in Figure 5.1, “Rails RESTful Routing” on page 54.

Figure 5.4. Form helpers use objects to determine the URL and method



REST Overkill

When we scaffold, we get this nice JSON api that we can use to work with our application. We can retrieve, modify, and even remove recipes from a simple REST web service, and we didn't have to write any of it.

However, it's not always necessary. If we decided we weren't handling JSON requests for this cookbook, our code in the controller could be a lot more compact. For example, our `create` action would go from this:

```
# POST /recipes
# POST /recipes.json
def create
  @recipe = Recipe.new(recipe_params)
  respond_to do |format|
    if @recipe.save
      format.html { redirect_to @recipe, notice: 'Recipe was successfully created.'
 }
      format.json { render action: 'show', status: :created, location: @recipe }
    else
      format.html { render action: 'new' }
      format.json { render json: @recipe.errors, status: :unprocessable_entity }
    end
  end
end
```

to this:

```
# POST /recipes
# POST /recipes.json
def create
  @recipe = Recipe.new(recipe_params)
  if @recipe.save
    redirect_to @recipe, notice: 'Recipe was successfully created.'
  else
    render action: 'new'
  end
end
```

Eventually, you'll be able to write both versions of your controllers without the use of scaffolding, so you'll be able to easily choose which path is right for you.

5.2. Cleaning up the Index page

The index page is pretty good for starters, but it displays all of the fields. It might be nice to show just the recipe name and when it was last updated instead of the ingredients and instructions. Remember, the list page is built using whatever fields you specified in your scaffold command.

The worst thing about this page's interface is that it requires Javascript to be enabled in order to delete records. While most people have JavaScript enabled these days, a feature so simplistic shouldn't require JavaScript to work. Look at this code:

```
<td><%= link_to 'Destroy', recipe, method: :delete, data: { confirm: 'Are you
 sure?' } %></td>
```

Rails follows REST conventions, meaning that deleting records should require a DELETE request method. Browsers only understand GET and POST requests; they don't understand DELETE requests so Rails has to fake it with a hidden form field called `_method` which gets interpreted on the server side. To hide this from the user and prevent its display in the URL, the default Rails scaffolding uses JavaScript to submit a hidden form. If you have JavaScript disabled, clicking the "destroy" link takes you to the "Show" page instead!

If we change `link_to` to `button_to`, like this:

```
<td><%= button_to 'Destroy', recipe, confirm: 'Are you sure?', method: :delete %></
td>
```

then the form is written on the page directly and we no longer require JavaScript.

Replace the contents of the page `views/recipes/index.erb.html` with the following code:

Example 5.15. code/04_scaffold_cleanup/cookbook/app/views/recipes/index.html.erb

```
<h1>Listing recipes</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Last Updated</th>
    <th colspan="2"> </th>
  </tr>

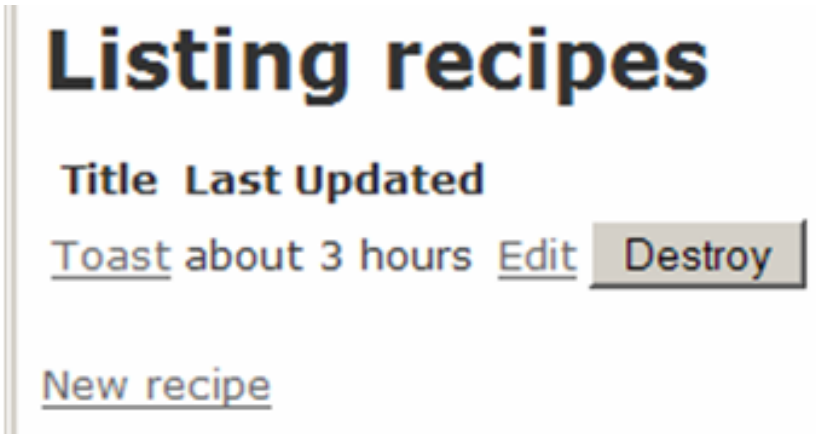
  <% @recipes.each do |recipe| %>
  <tr>
    <td><%= link_to recipe.title, recipe %></td>
    <td><%= time_ago_in_words(recipe.updated_at) %> ago</td>
    <td><%= link_to 'Edit', edit_recipe_path(recipe) %></td>
    <td><%= button_to 'Destroy', recipe, confirm: 'Are you sure?', method: :delete
 %></td>
  </tr>
  <% end %>
</table>

<%= link_to 'New Recipe', new_recipe_path %>
```

We've made the page look just a little nicer. We're using a new helper method here called `time_ago_in_words` which, when given a date, computes the difference and prints it out in words like “5 minutes ago” or “10 days ago.” This looks nicer than a long date and time stamp.

Refresh the index page in your browser. Your new page should resemble something like the one in Figure 5.5, “The modified Index page” on page 63.

Figure 5.5. The modified Index page



There are tons of other helper methods available for you to use. The Rails documentation on ActionView Helpers [1] has a lot more information. There you'll find some cool things like the `number_to_currency` helper which takes a number and formats it as dollars and cents.

[1] http://api.rubyonrails.org/classes/ActionView/Helpers.html

5.3. Cleaning up the Show page

Remember how the flow works in Rails? When we view a single recipe's detail page, the request is captured by Rails' router which sends the request to the `Recipes` controller's `show` action. But before that happens, there's a `before_action` filter that calls a method called `set_recipe` which uses the ID in the URL to fetch the record from the database:

Example 5.16. code/04_scaffold_cleanup/cookbook/app/controllers/recipes_controller.rb

```
# Use callbacks to share common setup or constraints between actions.
def set_recipe
  @recipe = Recipe.find(params[:id])
end
```

In this method we store the recipe we found into an instance variable; remember that the `@` means instance variable in Ruby. This instance variable is made available to the `show.html.erb` view. We can easily display any of the information about our recipe in our view. Right now the code on that page is just a bunch of paragraphs and it's not as organized as it could be.

Open `app/views/recipe/show.html.erb` and replace the contents with the following code:

Example 5.17. code/04_scaffold_cleanup/cookbook/app/views/recipes/show.html.erb

```
<article>
  <header>
    <h2>
      <%= @recipe.title %>
      [ <%= link_to 'Edit', edit_recipe_path(@recipe) %> ]
    </h2>
  </header>

  <h3>Ingredients:</h3>
  <p><%= @recipe.ingredients %></p>

  <h3>Instructions</h3>
  <p><%= @recipe.instructions %></p>
</article>

<p><%= link_to 'Back', recipes_path %></p>
```

Your page should look something like Figure 5.6, “The Show Page” on page 64

Figure 5.6. The Show Page

5.4. Improving our Layout

Our layout needs a few changes. Let's get some feedback going for the users when they create records successfully, and then let's add the new Recipes interface to our navigation bar.

User Feedback

When the user successfully creates, updates, or deletes records, we want to provide them with feedback, and Rails has a built in mechanism to do that called the Flash. We've been setting the Flash in our controller actions, but we're not displaying it on the pages anywhere. Our Show page had it at the top of the page, but we removed it because it really belongs in the layout so that we can use it wherever we want. Open up `app/views/layouts/application.html.erb` and add this code right below the navigation area:

Example 5.18. code/04_scaffold_cleanup/cookbook/app/views/layouts/application.html.erb

```
<% if notice %>
  <div id="notice">
    <%= notice %>
  </div>
<% end %>
```

Now we can display the notice, but only when it actually has a message to display.

Adding a Recipes link to the Navigation Bar

Adding the recipes interface to our site-wide navigation is easy enough. We can use one of the URL helpers we get for free with the resource routing.

```
$ bundle exec rake routes
```

shows us the routes we've defined so far, and according to its output, we can see that the prefix `recipes` points to the `index` action of the `recipes` controller.

```
     Prefix Verb   URI Pattern                Controller#Action
recipes GET    /recipes(.:format)          recipes#index
```

Thus, we add the suffix `_url` and place `recipes_url` in our navigation bar like this:

```
<%=link_to "Recipes", recipes_url %>
```

Add it to the navigation bar like this:

Example 5.19. code/04_scaffold_cleanup/cookbook/app/views/layouts/application.html.erb

```
<nav>
  <%=link_to "Home", root_url %>
  <%=link_to "Recipes", recipes_url %>
  <%=link_to "About", about_url %>
</nav>
```

Now you can easily get to the Recipes list no matter where you are.

5.5. Updating the Home Page

Our home page has a hard-coded list of recipes. Let's change that so it accurately reflects the number of recipes we have in our database. Open up `app/controllers/home_controller.rb` and change the `index` action so it looks like this:

Example 5.20. code/04_scaffold_cleanup/cookbook/app/controllers/home_controller.rb

```
def index
  @number_of_recipes = Recipe.count
end
```

The `count` calls the SQL statement `select count(*) from recipes` and gives us back the number of recipes in our system.

5.6. What You Learned

In this chapter we learned how a typical Rails application works, by observing how the scaffolding created a model, controller, and a set of views. We saw how data flows from a model, through the controller, and onto a view, and we learned how data comes from the user and ends up in our database.

We also took a little time to clean up some of the messes that the scaffolding made, but we also hooked up our recipes interface to our site's navigation, and learned how to place the notice messages in the application's template. This will come in handy later when we add new models to our app.

Exercises

1. Change all the links in the navigation bar from `link_to` to `link_to_unless_current`, a helper that makes the hyperlink active unless you're on the page it links to.

2. Alter the layout so the HTML `<title>` tag prints out a variable called @title. In each controller action in the Recipes controller, set @title to the title of the page.

   Then in `app/controllers/application_controller.rb` create a `before_action` that sets the `@title` variable to "Cookbook" so there's always a default value.

Now that the scaffolding is cleaner, let's look at how we can validate user input in our application, and how we can write some code that will test our app.

Chapter 6. Validating User Input

You may notice that if you entered recipes into the system without filling in any fields, the data was still placed into the database. But our next story on the list says that's not how it's supposed to work:

- As a site owner, I want to make sure that the title, instructions, and ingredients for a recipe are required fields so that a recipe is complete.

We can make certain fields required by using Rails' simple built-in validations. But we're going to go a step further than that and look at how we can use unit tests to ensure that our validations work.

In Rails, we do our data validation in the model. This is actually good practice because the models could be used outside of the Web. The MVC pattern standard is to place all business logic and business rules in the models and have only presentation logic in the views and controllers.

6.1. Rails Validations

Validations are special methods provided by the Validations feature in Active Record. There are quite a few built-in methods we can use to validate data. For simplicity, we'll use only `validates_presence_of` on our Recipe model.

Validation With One Line

Believe it or not, we can ensure that the user enters something for all three fields of our recipe with a single line of code.

Open `app/models/recipe.rb` and change the contents to

Example 6.1. code/05_validation/cookbook/app/models/recipe.rb

```
class Recipe < ActiveRecord::Base
  validates_presence_of :title, :ingredients, :instructions
end
```

Notice that we use *symbols* as the parameters to the `validates_presence_of` method. Remember that symbols are a special type of string used in Ruby to denote a value. For now, just think of them as immutable strings, or strings that cannot be changed and are used as labels in code. Ruby developers often use them as keys in hashes because a symbol used over and over in code references the same memory location, whereas a string used repeatedly is a unique object each time with its own memory allocated.

The `validates_presence_of` method is taking in an array of symbols which represent the attributes that need to be validated.

This simple line of code is all we need to make sure that users enter something for the title, ingredients, and instructions for a recipe. It’s not foolproof, but it’s a good start.

If you want to see what other types of validations are out there, take a look at this page in the Rails API: `http://api.rubyonrails.com/classes/ActiveRecord/Validations/ClassMethods.html`

6.2. Unit Tests

We need to put on the brakes here and write a quick unit test to ensure that our validation works. Unit tests are built right into the Rails framework and will become vitally important to your development process because they allow you to test your business logic and prove that things are working at the model and controller level. This way you don’t have to keep using a browser to test your app's logic. Best of all, as you get more comfortable writing tests, you'll spend less time tracking down bugs in the browser.

More importantly, once you understand how Rails works, you can begin using unit tests to design your code and drive the development of new features.

Write Tests First!

Normally, we'd have written our test cases before we implemented any code. This is one reason why scaffolding is bad; it discourages you from doing the right thing. However, since you're new to Rails, we're doing things a little bit differently.

Rails applications support unit testing right out of the box, and the process is automatic. Let's see how it works.

Your First Unit Tests

Rails automatically generated a unit test skeleton for recipe when we generated the recipe model. Open the file `test/models/recipe_test.rb` and change the contents to the following:

Example 6.2. code/05_validation/cookbook/test/models/recipe_test.rb

```
require 'test_helper'

class RecipeTest < ActiveSupport::TestCase
  test "creates a valid record" do
    recipe = Recipe.new
    recipe.title = "Ice water"
    recipe.ingredients = ["one glass","water","ice"].join("<br>")
    recipe.instructions = "Combine all ingredients into the glass and let sit for
 two minutes. Serve immediately."
    assert recipe.save
  end

  test "should not save unless title is filled in" do
    recipe = Recipe.new
    assert !recipe.save  # save should fail because there are errors.
    assert recipe.errors[:title].include?("can't be blank")
  end

  test "should not save unless ingredients is filled in" do
    recipe = Recipe.new
    assert !recipe.save  # save should fail because there are errors.
    assert recipe.errors[:ingredients].include?("can't be blank")
  end

  test "should not save unless instructions is filled in" do
    recipe = Recipe.new
    assert !recipe.save  # save should fail because there are errors.
    assert recipe.errors[:instructions].include?("can't be blank")
  end
end
```

We have four tests in this example. The first test creates a new instance of `Recipe`, sets the values for the recipe's title, instructions, and ingredients, and then saves the record. The `save` method returns `true` when there are no problems saving the record to our database. The `save` method calls the `validate` method automatically, which runs our validations. So we use the `assert` to evaluate the value of `save`. If the result of `save` evaluates to `true`, this test passes. If not, it fails.

This particular test needs to pass all the time, as it's the baseline test. If this test starts failing, that's an indication that the program's logic has changed somehow.

The other three tests simply attempt to save the record without setting one of the required fields. We expect these to fail because of our validations—in this test we haven't actually provided any of the required fields, but we are testing for only one error at a time to avoid making our tests too complicated. We're also asserting the inverse of true for the save. (assert that `Recipe.save` does not evaluate to true. Then we assert that the error messages are set for each field. Each validation has its own message format. In this case, the `validates_presence_of` validation stores "can't be blank" in the errors collection, under a key for each invalid attribute. If the `title` isn't blank, you'll find the error message for that in the errors collection, under the `:title` key.

How Tests Work With Data

Rails tests use the test database defined in `database.yml`. In our case, we'll have a test database that uses SQLite3. A test run starts by dumping out everything in

your test database and resetting it so it's completely empty. This ensures that you are always testing with the same data each time.

Keep Databases Separate

Never use the same database for production, development, and testing!!!!

Each test is independent of the others and the database's state is reset after each test runs. You can feel free to add and delete as many records as you want in a test and they will be recreated when you start the test again. Of course, you may want to actually have some data in your database to test with. In order to have existing records in your tests, you need to create them somehow. You can use pure Ruby code to create new records in your tests, as we've done so far, but the default way to seed your test database with dummy data is to use Fixtures.

Fixtures

Tests can get data from fixtures. Fixtures are loaded into each test by the fixtures method. You'll need a fixture for each table in your database, not each model in your system.

Modify the fixture for the recipes table by editing `/test/fixtures/recipes.yml` and add a few recipes.

Formatting is Important

Be careful not to use tabs and also be sure to leave a space after each colon! YAML is a tricky little format.

Example 6.3. code/05_validation/cookbook/test/fixtures/recipes.yml

```
# Read about fixtures at http://api.rubyonrails.org/classes/ActiveRecord/
FixtureSet.html
ice_water:
  title: "Ice Cream"
  ingredients: "3 scoops vanilla ice cream<br>chocolate syrup"
  instructions: "Scoop ice cream into the bowl and pour chocolate syrup on top."
toast:
  title: "Toast"
  ingredients: "bread, butter, jelly"
  instructions: "Place bread in the toaster for 1 minute. Remove from toaster and
 apply butter to each piece."
```

When the test is run, this data gets loaded into the recipes table and is available within the test. The tests you currently have in your test don't need these fixtures, but you may write future tests that depend on having data in the test database. For example, you may want to write a test to make sure that there can only be one recipe called "Toast". That test might look something like this:

```
test "should only have one recipe with the same name" do
  recipe = Recipe.new(:title =>"Toast")
  recipe.valid?
  assert recipe.errors[:title].include?("must be unique")
end
```

Running the Test

To run the test, we need to first prepare the test database. We do that by running a rake task.

```
$ rake db:test:prepare
```

This task takes the structures from our development database and creates a new test database that we can use over and over again.

Our test is actually a standalone Ruby application. We can run the test directly using Ruby.

```
$ bundle exec ruby -Itest test/models/recipe_test.rb
```[1]

Everything should work well. You should get no errors or failures, expressed on your screen like this:

```
        Run options: --seed 59345

# Running tests:

....

Finished tests in 0.066490s, 60.1594 tests/s, 105.2790 assertions/s.
```

Each dot you see is a successful assertion. If you see E or F instead of a dot, you've got an error or a failure instead and you'll need to look at the messages the test report gives you.

You can run all of the unit tests by running

```
$ bundle exec rake test:units
```.

This also takes care of initializing the test database for you. This is the preferred way of running all of your tests; as your application grows, you'll want to run all of your tests, not just a single file.

6.3. Providing Feedback to Users

Now that we know our validation works, we should see what it looks like when users attempt to leave fields blank. Visit the “New Recipe” page, leave all of the fields blank, and press the “Create” button.

[1]The -Itest argument tells Ruby what folder it should look in to load additional files. Our test case above requires a file called 'test_helper' which is located in the `test` folder. Telling the Ruby interpreter that the `test` folder should be loaded up makes our test work.

Recall how the cycle works. Our data is submitted to the `create` method of `RecipeController`:

```
# POST /recipes
# POST /recipes.json
def create
  @recipe = Recipe.new(recipe_params)
  respond_to do |format|
    if @recipe.save
      format.html { redirect_to @recipe, notice: 'Recipe was successfully created.'
 }
      format.json { render action: 'show', status: :created, location: @recipe }
    else
      format.html { render action: 'new' }
      format.json { render json: @recipe.errors, status: :unprocessable_entity }
    end
  end
end
```

That method creates a new instance of a `Recipe`, passing in the data from our form. Then it calls the `save` method on the `@recipe` instance. The `save` method will return false because it runs our validations.

Active Record's Validations places error messages in an collection on the instance of the model called `errors`. So, the controller renders the "New Recipe" form, and the form redisplays on the screen. The form partial used on the "New Recipe" form contains code that reads the error messages and displays them to the screen:

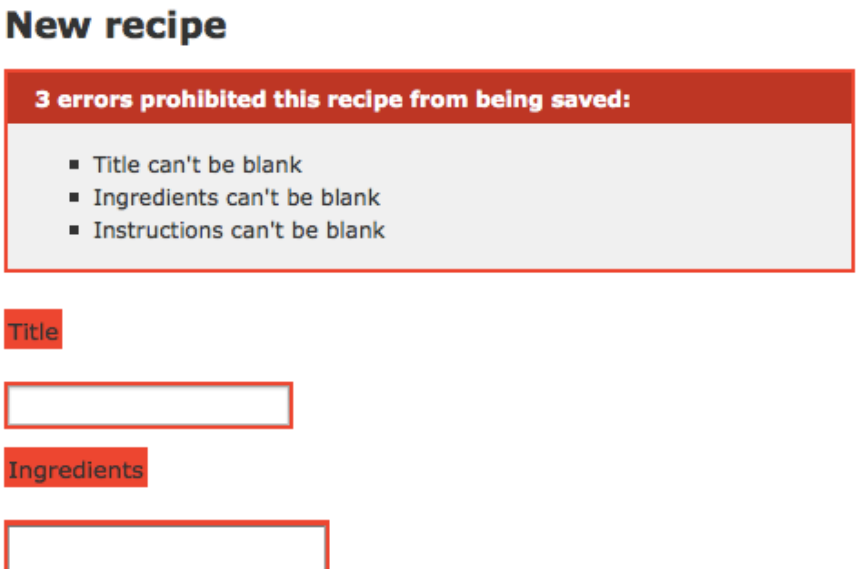
```
<% if @recipe.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@recipe.errors.count, "error") %> prohibited this recipe from
 being saved:</h2>

    <ul>
    <% @recipe.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

Normally, this code doesn't do anything. But since the `@recipe` object has error messages, they'll appear on the screen.

Built in helper methods for text boxes, select boxes, and text areas are validation-aware. They will become styled automatically, providing additional visual cues. The styles are applied using CSS, so you can modify the way they look. Take a look at Figure 6.1, "User Feedback as provided by Rails validations" on page 73 to see the results.

Figure 6.1. User Feedback as provided by Rails validations



This task alone could take a web developer a few hours to get right. We've created a working solution with a unit test in only a few minutes. These validations work for new entries and existing entries.

6.4. What You Learned

In this chapter you learned about how Rails does unit testing and how you can use Rails' built-in validations to keep users from entering junk into your database.

Exercises

1. Write the following unit test in `test/models/recipe_test.rb`

```
test "should only have one recipe with the same name" do
  @recipe = Recipe.new(:title =>"Toast")
  @recipe.valid?
  assert @recipe.errors.on(:title).include?("must be unique")
end
```

   Run the unit tests for the application and ensure that this new test fails. (It should fail, because we already have a Fixture that includes a recipe called "Toast.") Then add a uniqueness validation to the recipe title field. See the Rails documentation for how to do this.

2. Make it apparent that the three existing fields on the form are required fields.

We've got a pretty good application here. We can add recipes and other people can view them. Let's take it further now and add a second model to our application so we can categorize recipes.

Chapter 7. Adding Categories

So far, we've worked with a single database table, but it's more common to work with multiple tables of data in a real world web application. In this chapter we'll explore how to create database relationships.

Let's look at this user story:

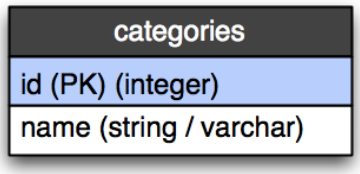
- As a site owner, I want to be able to place recipes in categories so I can organize them better.

For this example, we'll just say that a recipe can only belong to one category, and a category can have many recipes. We'll say it that way because that's how Active Record allows us to express relationships.

7.1. Creating a category model and table

We don't need to create a full interface to manage categories at this time. We just need to create a model.

Figure 7.1. The Categories table



The `Category` model maps to a 'categories' table and assumes the table has a primary key called ID. Notice how Rails handles the pluralization correctly!

We used the scaffold generator before, but Rails has generators that just create the model. We create the new model by dropping to a command prompt and typing

```
$ rails generate model category name:string
```

You'll see something similar to this in your terminal:

```
invoke  active_record
create    db/migrate/20140629143806_create_categories.rb
create    app/models/category.rb
invoke    test_unit
create      test/unit/category_test.rb
create      test/fixtures/categories.yml
```

Run the newly-created migration by executing the command

```
$ bundle exec rake db:migrate
```

from the command line. It will create the new table.

```
==  CreateCategories: migrating ===============================================
      -- create_table(:categories)
         -> 0.0014s
      ==  CreateCategories: migrated (0.0015s)
 ======================================
```

With that, we can start populating our database table with categories.

7.2. Adding some default records with Rake

Sometimes it's nice to have your database pre-populated with records. You saw how fixtures can do that with test data, but that's not always a good choice. Migrations could be used to insert data into your database but that can be volatile as well. The best approach is to use rake, which is the same tool you've been using to run your migrations.

Rake is an automation language. To use it, you simply create a file with some tasks and then execute it via the rake command.

Rails projects look for Rake tasks in files with the .rake extension in the project's `lib/tasks` folder. Create a new file in that folder called `import.rake`. Place this code in the file:

Example 7.1. code/06_category/cookbook/lib/tasks/import.rake

```
namespace :db do
  desc "Puts default categories in the database"
  task :import_categories => :environment do
    Category.create :name =>"Beverages"
    Category.create :name =>"Deserts"
    Category.create :name =>"Appetizers"
    Category.create :name =>"Entrees"
    Category.create :name =>"Breakfast"
    Category.create :name =>"Sandwiches"
  end
end
```

A *namespace* is just a container for code like in any other language. When you issued the command `bundle exec rake db:migrate` you called the migrate task within the db namespace. We'll follow that same convention here.

To import the records into your database ,issue the command

```
$ bundle exec rake db:import_categories
```

seed.rb

The approach we just took shows you how to make your own custom Rake tasks for your application. But Rails includes a feature that lets us specify the records that should exist in the database by default. In the file `db/seeds.rb` we can write the

same record-creation code we wrote for the Rake task, but without the task definition:

Example 7.2. code/06_category/cookbook/db/seeds.rb

```
# categories
Category.create :name =>"Beverages"
Category.create :name =>"Deserts"
Category.create :name =>"Appetizers"
Category.create :name =>"Entrees"
Category.create :name =>"Breakfast"
Category.create :name =>"Sandwiches"
```

Then when we want to run this, we just use

```
$ bundle exec rake db:seed
```

A major advantage of this is that we can run the task

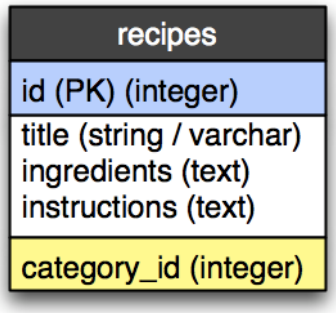
```
$ bundle exec rake db:setup
```

to create or even re-create) and seed the database with records in a single command, which works out great when we want to deploy our system to it's production database.

7.3. Modifying the Recipes table

Since we want to have a relationship between categories and recipes, we have to place a foreign key in the recipes table so it can be associated with a recipe. We'll do this with a migration too.

Figure 7.2. Recipes table with the category_id column



Adding the foreign key to the Recipes table prepares us to use Rails to handle the relationships.

We'll create a new migration to alter our database. From the command line, execute the command

```
$ rails generate migration add_category_id_to_recipes
category:references
```

This creates a new migration file `db/migrate/XXXXXXX_add_category_id_to_recipes.rb`. Open the file and you should see the following code:

Example 7.3. code/06_category/cookbook/db/migrate/20140629144357_add_category_id_to_recipes.rb

```
class AddCategoryIdToRecipes < ActiveRecord::Migration
  def change
    add_reference :recipes, :category, index: true
  end
end
```

The generator actually added the column we needed to the Recipes table! It saw that the migration's name started with `add` and then parsed out the `to_recipes` part of the migration name to figure out the table. Then it saw that we wanted to create a "reference" to a "category", and created a `category_id` column on that table. Even better, it created an index on the column!

Rails doesn't support database-based foreign key constraints because not every database supports them, and the Rails creator felt that the logic for controlling constraints should live in the application, not the database. However, even though there's no constraint, we still should have an index on the column to improve performance.

Run the migration to alter the database. (`$ bundle exec rake db:migrate`).

```
==  AddCategoryIdToRecipes: migrating =========================================
 -- add_reference(:recipes, :category, {:index=>true})
    -> 0.0098s
==  AddCategoryIdToRecipes: migrated (0.0099s) ================================
```

If you don't see changes

In some rare cases, changes to your application won't work without restarting the development server. Press CTRL+C (or CTRL+Break on Windows) to stop your web server and then restart it by executing the command

```
$ rails server
```

7.4. Creating an Association Between a Recipe and a Category

Associations allow objects to interact. Associations are methods that map the primary keys of one table to the foreign keys of another; the relational mapping part of “object-relational mapping”.

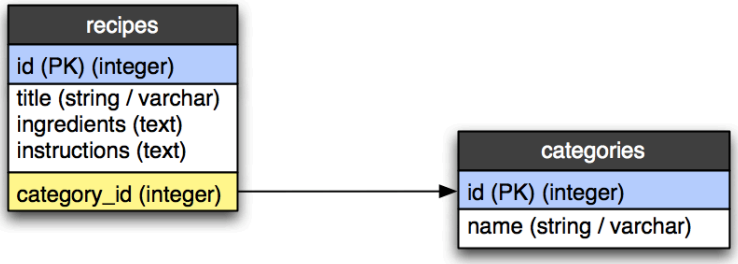
Open `app/models/recipe.rb` and modify its contents with the following code:

Example 7.4. code/06_category/cookbook/app/models/recipe.rb

```
class Recipe < ActiveRecord::Base
  belongs_to :category
  validates_presence_of :title, :ingredients, :instructions
end
```

The `belongs_to` class method takes in a symbol name of a class with which we wish to associate. Rails needs no further information because it will assume that `:category` references a class called `Category`, that the table name will be `categories`, and that this table (`recipes`) has a foreign key column called `category_id` that will reference the id column in the `categories` table. Of course, we can override these assumptions, but it's easier just to follow convention.

Figure 7.3. Recipe Belongs To a Category



Rails uses naming conventions to discover relationships. The Recipe model gets the belongs_to directive because it has the foreign key category_id.

This association adds some new methods to an instance of `Recipe`. For example, we can retreive the category associated with a recipe with the `category` method. And that means that we can use dot chaining to access the name of the category directly through a method on the recipe object:

```
recipe = Recipe.find(1)   # gets recipe with id of 1
      recipe.category.name  # gets the associated category name
```

However, this dot chaining can be a little dangerous. If the recipe has no category assigned, we'll get an error. It's best to get the category from the recipe, then only grab the category name with the `name` method if the `category` object isn't Nil. We'll look at how to do that shortly because we're going to want to display a recipe's category. But first we'll have to modify our application so we can actually associate categories with our recipes.

7.5. Adding Categories to the Forms

When we add or update a Recipe, it would be nice if we had a dropdown list that would let us associate a category with that recipe. This is remarkably easy to do in Rails. In fact, it's many times easier than it is in something like PHP. But in order to add the category selection to the forms, we need to do some work in the controller.

Open `app/controller/recipes_controller.rb` and locate the `new` method. We need to modify it so it retrieves the categories into an instance variable called `@categories` which we can then use in our view.

Example 7.5. code/06_category/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes/new
def new
  @recipe = Recipe.new
  @categories = Category.all.collect{|c| [c.name, c.id] }
end
```

Now find the `edit` method and modify it so it also retrieves the categories into `@categories`.

Example 7.6. code/06_category/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes/1/edit
def edit
  @categories = Category.all.collect{|c| [c.name, c.id] }
end
```

Array Collection

The `collect` method, and its alias `map`, iterates through all the categories and returns a new array containing the elements we're specifying. This is an example of a block and you'll see lots of these in Rails applications and Ruby code.

In this case we're returning an array of arrays containing the name and ID. We'll use the `select` helper method in our view to build the dropdown list of categories, and it can take this type of array.

Open `app/views/recipes/_form.html.erb` and add the following block at the end of the form:

Example 7.7. code/06_category/cookbook/app/views/recipes/_form.html.erb

```
<div class="field">
  <%= f.label :category_id %><br />
  <%= f.select :category_id, @categories, :include_blank => true %>
</div>
```

We're adding a select box which will contain all of the categories we collected so that our users can place a recipe into the category chosen by the dropdown. We're also including a blank option so that a user doesn't have a category already selected when they view the page. The category name is displayed on the screen, but the actual value that's passed on the form is the ID of the category, which is perfect as this select box is bound to the `category_id` field.

Allowing Parameters

The Scaffold generator created a special method that determines what parameters we're allowed to send from our web forms. Right now it allows us to send the `title`, `ingredients`, and `instructions` to the controller. But we've now

added a select field that sends `category_id`. As a security measure, this parameter is silently dropped and never sent on to the database. We have to alter the generated `recipe_params` method to allow the `category_id` field. If we don't, then no matter what the user chooses in the dropdown list on the page, their changes won't be saved to the database. Here's what the `recipe_params` should look like with that change:

Example 7.8. code/06_category/cookbook/app/controllers/recipes_controller.rb

```
# Never trust parameters from the scary internet, only allow the white list
 through.
def recipe_params
  params.require(:recipe).permit(:title, :ingredients, :instructions, :category_id)
end
```

Any time you add a new field to the form, you have to add it to this method. This prevents people from attempting to modify your database through web requests by letting you whitelist the fields you want to expose to the world.

With the associations in place and the views fixed up, go ahead and play with your application. Notice how you can now add categories to your recipes, and when you edit an existing recipe, its associated category automatically shows up in the dropdown list. But now let's make sure the category shows up on the Show and Index pages of our app.

7.6. Displaying Records and Associations

When we bring up the index page, we'll want to show the recipes and their categories, and when we click on a single recipe we'll want to see the category name associated there as well. Thanks to the associations we've defined in our models, this is a piece of cake. But before we go about changing the views, let's look at how Rails writes queries for databases. We'll need to know how that works because it can impact the performance of our app.

Lazy vs. Eager Loading

As mentioned previous in Section 7.3, “Modifying the Recipes table” on page 77, when we add associations to models we get some extra methods that let us grab the associated records. Take another look at this code:

```
recipe = Recipe.find(1)   # gets recipe with id of 1
        recipe.category.name  # gets the associated category name
```

The preceding code will fetch the data from the database using*lazy loading*, meaning that it'll grab the Recipe data with one query, and then grab the Category record with a second query. It doesn't do an explicit or an implicit `JOIN` on the tables. Let's explore this with a tool called the Rails Console.

Open a new Terminal window (or command prompt) and navigate to the root of your Rails project. You could stop the Rails Server and use the same Terminal win-

dow if you wanted, but then you'd have to remember to start up the server when we're done. Most Rails developers are comfortable working with multiple Terminal windows all pointing to the same Rails project.

Open the Rails console with the command

```
$ rails console
```

You'll see something similar to this:

```
Loading development environment (Rails 4.1.2)
  :001 >
```

When you see this message, it means that your Rails application has been loaded up and you now have access to your models and other components of Rails. We can use this to see exactly how Rails talks to our database.

In the console, type the following statement:

```
recipe = Recipe.find(1)
```

This returns the following result:

```
D, [2013-12-07T12:13:53.547318 #18584] DEBUG -- :   Recipe Load (8.5ms)  SELECT
 "recipes".* FROM "recipes" WHERE "recipes"."id" = ? LIMIT 1  [["id", 1]]
  => #<Recipe id: 1, title: "Toast", ingredients: "Bread", instructions: "Put bread
 in toaster", created_at: "2013-11-17 19:22:22", updated_at: "2013-11-17 19:23:12",
 category_id: 5>
```

Now type

```
recipe.category
```

and you'll see that a second query gets generated:

```
D, [2013-12-07T13:23:24.912382 #18705] DEBUG -- :   Category Load (0.7ms)
 SELECT "categories".* FROM "categories" WHERE "categories"."id" = ? ORDER BY
 "categories"."id" ASC LIMIT 1  [["id", 5]]
=> "Breakfast"
```

Rails tries to be efficient, and so it only makes queries when it has to. But this could be bad if we were retrieving all recipes and displaying the category for each one. If you have 200 recipes and wanted to display the category for each one, like this:

```
recipes = Recipe.all
recipes.each do |recipe|
  recipe.category.name
end
```

then the preceding code would generate 201 SQL statements! One statement for the initial recipe table load, and one statement to fetch the category for each entry in the recipes table!

```
 D, [2013-12-07T13:33:04.926417 #18757] DEBUG -- :   Recipe Load (0.2ms)  SELECT
"recipes".* FROM "recipes"
           D, [2013-12-07T13:33:04.928045 #18757] DEBUG -- :   Category Load
(0.2ms)  SELECT "categories".* FROM "categories" WHERE "categories"."id" = ? ORDER
BY "categories"."id" ASC LIMIT 1  [["id", 5]]
           D, [2013-12-07T13:33:04.929087 #18757] DEBUG -- :   Category Load
(0.1ms)  SELECT "categories".* FROM "categories" WHERE "categories"."id" = ? ORDER
BY "categories"."id" ASC LIMIT 1  [["id", 5]]
....
```

That's gross, and it'll kill our database server's performance if we put this application in front of more than a couple of users. Thankfully there is a solution for situations like this called eager loading. Rails will eager load things for us if we specify the objects to include.

```
recipe = Recipe.all
```

becomes

```
Recipe.includes(:category)
```

This reduces the number of calls to the database to two calls: one call to fetch the recipes, and another to fetch the categories. Behind the scenes, Rails matches the category to the record, eliminating the need for additional queries.

You could have Rails generate a proper `LEFT JOIN` by adding the `references` method:

```
Recipe.includes(:category).references(:category)
```

But beware of this because your result from the database may have many characters. In our case the categories table has a single field. But if it had many fields, this approach might not be as performant as letting Rails handle it.

Finally, if you want to fetch a single recipe and its category using a single query with a `LEFT JOIN`, then you just search for the specific record by its ID with `find`:

```
Recipe.includes(:category).references(:category).find(1)
```

If you run each of these examples from the Rails console, you'll be able to see the SQL statements that Rails generates. You can also see these statements in `log/development.log`. You should periodically review the SQL statements that Rails creates for you because you may need to tweak them to improve performance in your application. Just because it runs fast on your development machine with a single user doesn't mean it'll perform the same with hundreds or thousands of users hitting it during peak times.

So now that you know how Rails creates SQL statements for associated records, let's put it to use by fixing up our “show” and “index” pages to display the category associated with each recipe.

Adding the Category to the Show view

When we display a recipe, we want to now show the category for the recipe. We can do that easily thanks to the way the `belongs_to` association works. When we associated a category to a recipe using that association, it added a method to our Recipe model called `category`, which returns an instance of the associated category record.

Locate the `set_recipe` method in `recipes_controller` and add eager loading for the category using `where` and `include`

Example 7.9. code/06_category/cookbook/app/controllers/recipes_controller.rb

```
# Use callbacks to share common setup or constraints between actions.
def set_recipe
  @recipe = Recipe.includes(:category).find(params[:id])
end
```

Open `app/views/recipes/show.html.erb` and add

Example 7.10. code/06_category/cookbook/app/views/recipes/show.html.erb

```
<p>Category: <%= h(@recipe.category.name) rescue "No category found" %></p>
```

somewhere on the page. When you refresh, you'll see the category displayed.

Rescuing Exceptions

The `rescue` statement catches a possible exception that could be thrown if the recipe does not yet have an assigned category. Your recipes don't all have the category assigned yet, and without this `rescue` statement, this page would fail to render. Be careful of inline rescues though, because they can swallow errors and make it much harder for you to debug problems.

Displaying Categories on the Index view

Let's add the category to our main list. Find the `index` action in `recipes_controller.rb` and add the `:include` option to the `find` to eager-load the category information just like the previous example.

Example 7.11. code/06_category/cookbook/app/controllers/recipes_controller.rb

```
# GET /recipes
# GET /recipes.json
def index
  @recipes = Recipe.all
end
```

Open `app/views/recipes/index.html.erb` and modify it so we can see the category name in the table. We'll need to add a column heading as well as the

data cell itself. Remember to use the association to retrieve the category name just like you did on the show page!

Example 7.12. code/06_category/cookbook/app/views/recipes/index.html.erb

```
<h1>Listing recipes</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Category</th>
    <th>Last Updated</th>
    <th colspan="2"> </th>
  </tr>

  <% @recipes.each do |recipe| %>
  <tr>
    <td><%= link_to recipe.title, recipe %></td>
    <td><%= recipe.category.name rescue "No category" %></td>
    <td><%= time_ago_in_words(recipe.updated_at) %> ago</td>
    <td><%= link_to 'Edit', edit_recipe_path(recipe) %></td>
    <td><%= button_to 'Destroy', recipe, confirm: 'Are you sure?', method: :delete
 %></td>
  </tr>
  <% end %>
</table>

<%= link_to 'New Recipe', new_recipe_path %>
```

That wasn't too bad, was it?

7.7. Before Filters

There's a slight problem with our cookbook application. If we attempt to create a recipe that doesn't validate, we now get an error when redisplaying the "New Recipe" page. This is because while we've initialized the list of categories for the dropdown list in the `new` action, we haven't done so in the `create` action. When recipe creation fails, the "New Recipe" template is rendered, but the `new` action is not called.

We're already duplicating code in both the `new` and `edit` actions, and we don't want to duplicate it again in the `create` and `update` actions. Thankfully with Rails we don't have to, as there's a mechanism we can use to execute code at the beginning of each request. It's called a “Before Filter”, and if you recall, we've already seen this work when we looked at the scaffolding.

Add this code to `app/controllers/recipes_controller.rb`, right after the class definition:

Example 7.13. code/07_before_filters/cookbook/app/controllers/recipes_controller.rb

```
before_action :get_categories, :only =>[:new, :edit, :create, :update]
```

Before filters call methods before actions. In this case, we're calling the `get_categories` method before the `new`, `create`, `create`, and `update` actions.

before_filter

In previous versions of Rails, the method `before_filter` did the same thing as `before_action`. You should use `before_action` going forward as `before_filter` is being phased out.

At the bottom of the file, add this code, right above the very last `end` line.

Example 7.14. /Users/brianhogan/books/cookbook2/code/07_before_filters/cookbook/app/controllers/recipes_controller.rb

```
def get_categories
  @categories = Category.all.collect{|c| [c.name, c.id] }
end
```

We're just using the same code we placed in the `new` and `edit` actions into this filter. Instance variables you set in filters are also available in the actions and in the views. Now you should remove the category collection code from the `new` and `edit` actions before you move on.

One last thing about filters. We've used `before_action` to make code run before the user visits a controller's action, but Rails also supports `after_action` so you can make some code run after an action.

Now, open the app in the browser, locate the first record, choose the “Edit” button, and add a category by selecting the category from the dropdown box.

7.8. What You Learned

We covered a ton in this chapter. You got a chance to see how to add a second database table to your app and how to make relationships between the new table and your existing one. You also got to interact with the Rails Console and see how Rails creates queries. You got a chance to create a Rake script that creates several records in your database, and you modified the existing views to integrate the second table into the system.

Exercises

1. Use a database migration to add a new integer column to the Recipes table called “calories” and then add this new field to the forms and the Show page.

2. Create a `has_many` association from Category to :recipes

   - Write a unit test that tests this relationship. (You’ll need to make fixtures for categories and recipes and you’ll need to load both of these in the test file.)

   - See http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html for details on has_many

3. Create a controller and views to manage the categories. Use the recipe controller and views as an example. Do not use scaffolding, as this will overwrite things you've already done.

   - When you display a category, display the recipes associated with that category, making use of the has_many association you created. You should be able to retrieve the categories with

     ```
     @recipes = @category.recipes
     ```

     and then use the `each` method on the `@recipes` object.

   - On the Show page for a recipe, make a link to the category using the `link_to` method. Remember that the `link_to` method can take an object as its second parameter.

Now that we've explored how to join two tables together, and we've spent time querying our database with Active Record, we've got a nice basic appliation. But before we put it out for the world to see we'd better put some security in place. After all, right now anyone can add or delete recipes from our database!

Chapter 8. Security

Right now, anyone can add recipes to our cookbook. But according to the next story on our list, that's not we want:

- As a site owner, I want to have to log in to the site be able to create, update, or delete recipes, so that the general public can't modify my database and mess with my records.

So let's add some simple security to the system.

Rails provides built-in support for basic authentication, which is a great way to lock people out of your site. We want to authenticate users on each request, and we can leverage Rails' `before_action` feature to make that work.

8.1. Using Global Methods In The Application Controller

By default, all of our controllers inherit from `ApplicationController`. Any methods we place in that controller are all available in the rest of our controllers. This is a great place for us to put our filter methods that our controllers will share, like authentication.

Let's add a method prompts the visitor for a username and password when they visit certain actions on our site. Since we'll use this everywhere in our app, we'll add this code to `app/controllers/application_controller.rb`:

Example 8.1. code/07_before_filters/cookbook/app/controllers/application_controller.rb

```
private

  def authenticate
     authenticate_or_request_with_http_basic do |user_name, password|
       session[:logged_in] = (user_name == 'admin' && password == 'password')
     end
  end
```

This method uses HTTP Basic Authentication. It's not very secure since it uses a single username and password for everything, but it's a great first step towards securing our application and exploring how Basic Authentication works.

HTTP Basic Authentication is incredibly simple. When you make the request from your browser, the remote server asks you for credentials. You enter the username and password in and the browser takes those credentials and joins them together like this:

```
username:password
```

It then encodes the username and password using simple Base64 encoding and sends this to the server in the `Authorization` header. The browser caches these cre-

dentials and sends them along any time the server requests them. This way the user isn't constantly prompted for their username and password.

The username and password isn't encrypted, and so in production, Basic Authentication, like other types of authentication, should be used with SSL.Setting that up is beyond the scope of this tutorial.

Sessions

Every connection to a web server over HTTP, the protocol of the web, is a single, short connection. The client makes a request to a server, the server sends a response back, and then the connection is terminated. However, we have some tricks we can use to make it look like we're always connected. One of those methods is called a session.

We can persist data in the session on one part of our site and then retrieve it later. It's great for storing small bits of data that we don't constantly want to look up from a database, but it's also used to keep track of whether or not someone's logged in.

In our `authenticate` method, we're using the session to store a boolean value that determines whether our user is logged in or not. We'll use this later to hide or show user interface elements in our view.

By default, Rails stores session data on the end-user's computer in a special encrypted cookie. There's a layer of security in the Rails application that makes it nearly impossible for someone to tamper with the cookie. This mechanism has undergone incredible scrutiny and has held up even under some of the harshest security tests. It's safe to use it, but you can also choose to store session information in a database on the server instead.

We've only declared the filter, we haven't applied it yet, so let's do that now.

8.2. Securing Destructive Actions

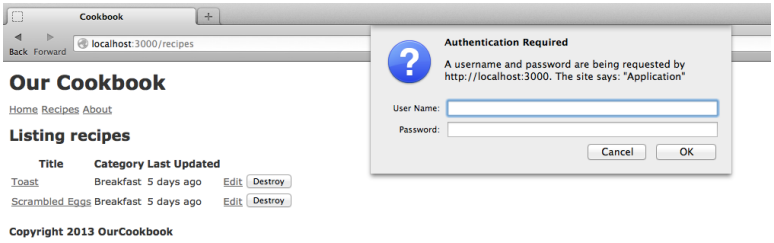
In our Recipes controller, we don't want unauthorized users to see anything except for the `index` or `show` pages. So, we can add this filter to `app/controllers/recipes_controller.rb`:

Example 8.2. code/07_before_filters/cookbook/app/controllers/recipes_controller.rb

```
before_action :authenticate, :except => [:index, :show]
```

Now, try deleting one of your recipes. You'll be prompted to enter the username and password.

Figure 8.1. Authentication Prompt



8.3. Hiding things people shouldn't see

People who aren't logged in shouldn't see links to create, edit, or delete posts. We can create a new helper method that we can use in our views to hide those elements. Add this to `app/controllers/recipes_controller.rb`:

Example 8.3. code/07_before_filters/cookbook/app/controllers/application_controller.rb

```
helper_method :logged_in?

def logged_in?
  session[:logged_in]
end
```

This method simply checks the value we set in the session hash. Now we can start hiding elements for users who aren't logged in. Let's start by hiding the "New recipe" link on our Recipes index page:

Example 8.4. code/07_before_filters/cookbook/app/views/recipes/index.html.erb

```
<% if logged_in? %>
  <%= link_to 'New recipe', new_recipe_path %>
<% end %>
```

You can now go through and use this technique to hide other elements on the index page, but be careful, because if you hide all of them, you won't have any way to log in to the site without typing the URL to a protected action like http://localhost:3000/recipes/new.

8.4. Limitations of This Authentication Method

We have used a very, very simple way of protecting the site. It relies on a hard-coded password and only supports one login. In addition, this method is not secure unless you use SSL certificates and use the HTTPS protocol. You also don't have any way to log out without closing the browser, since the browser is designed to cache credentials for basic authentication.

There are many alternatives you can use in your real applications, including Devise[1], which is extremely robust, but more complicated. It includes support for password recovery, account activation emails, and encryption.

8.5. What You Learned

In this chapter you learned about Basic Authentication and how to use it to restrict access to parts of your Rails application. You learned about the session and how to make helper methods that you can use in your view.

Exercises

In these exercises, you'll change your application so it uses a database table of usernames and passwords instead of a hard-coded username and password, and you'll create an admin dashboard page.

1. Create a users model in your database with the following attributes:
   - username (string)
   - password_digest (string)
2. Add this code to your `User` model in `app/models/user.rb`:

```
has_secure_password

def self.authenticate(username, password)
  user = User.where(username: username).first
  user && user.authenticate(password)
end
```

3. Add the `bcrypt` gem to your `Gemfile`:

```
gem 'bcrypt-ruby', '~> 3.1.7'
```

4. Create a new Admin user in your database.
   - From the Rails Console, add a user called `admin` with the password `password` like this:

```
User.create :username => "admin", :password
 => "password", :password_confirmation =>  "password"
```

     Then exit the Rails console by typing `exit` to return to your Terminal.
   - Add the same code you ran in the console to the file `db/seeds.rb` so that you can have a default user when you deploy the application to a new server.
5. Change the `authenticate` method in `app/controllers/application_controller.rb` so that it uses the `authen-`

[1] http://github.com/plataformatec/devise

`ticate` method we just added to `User`. The new `authenticate` should look like this:

```
if user = authenticate_with_http_basic {|user, password| User.authenticate(user,
 password)}
  session[:user] = user.id
  session[logged_in] = true
 else
  request_http_basic_authentication
 end
```

6. Add a new private method to `ApplicationController` called `current_user` that fetches the current user from the database using the data we've stored in the session:

```
def current_user
  @current_user ||= User.find(session[:user_id]) if session[:user_id]
end
```

   This method looks up the user and stores the user in the `@current_user` variable only if the `@current_user` variable hasn't yet been set, *and only if there's data in the session*.

   Then add a `helper_method` declaration to the top of `ApplicationController` for the `current_user` method you just defined so that we can use `current_user` in the view.

   Then, in your views, you can use this method to quickly load up the current user's information.

```
<% if logged_in? %>
  You are logged in as <%= current_user.username %>
<% end %>
```

7. Let's make it easy for users to log into our system. When they visit the URL http://localhost:3000/admin they'll be prompted for the login box and then greeted with a welcome message. Add a new controller to your system called `AdminController` with a single action called `index` and a single view called `index.html.erb`.

   - Ensure that the `AdminController` contains the code that authenticates users, but make sure that it has no exceptions.
   - On the `index.html.erb` page, simply present a message that states they are logged in.
   - Create a route in `config/routes.rb` that connects `/admin` to the `admin` controller's `index` action.

8. Finally, close your browser and reload it. Then visit http://localhost:3000/admin and log in with your new username and password.

Chapter 9. Managing Styles and Other Assets

We've one a lot of work with databases and static pages, but our application doesn't exactly look the best. We'll want to spend a little time making our application look nice now that things work. Rails provides an incredibly powerful system of managing CSS and JavaScript files called the Asset Pipeline, and we'll explore it as we create some basic stylesheets.

9.1. How Rails Handles Assets

Rails apps have a folder called `app/assets` that is designed to hold our CSS, JavaScript files, and our images. For simplicity, we'll only focus on the stylesheets right now.

Take a look at the `app/views/layouts/application.html.erb` file; you'll find this line:

Example 9.1. code/08_assets/cookbook/app/views/layouts/application.html.erb

```
<%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" =>
 true %>
```

Normally, you use a `link` tag in HTML and point it at the CSS file you want to load. But that often involves worrying about what folder your pages are in, and what folder the CSS file is in, and resolving paths. It's messy work, and so Rails takes care of this for us. The `stylesheet_link_tag` helper will, by default, simply load up the file we specify by looking for it in the `app/assets` folder. It will construct the appropriate relative link for us no matter what page we're on.

By default, it looks for the file `app/assets/stylesheets/application.css`. But if you look at that file you'll see this:

Example 9.2. code/08_assets/cookbook/app/assets/stylesheets/application.css

```
/*
 * 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 .
 */
```

You could start adding styles directly to this file, but that's not the approach Rails wants you to use. This file is actually a “manifest” file which contains

some special instructions that actually load up all of the files in the `app/assets/stylesheets` folder.

The Asset Pipeline uses this manifest to construct a single stylesheet file for production use. This lets us keep our stylesheets organized by function or any other method we choose, so that we can easily manage our code. Then, Rails assembles all of them into a single file, which is often more efficient for end users to download. If we made the user download several stylesheet files, the experience might be much slower for them due to the overhead caused by fetching the individual stylesheet files. Combining multiple stylesheets into a single one for production is an industry standard, and Rails includes it out of the box.

Preprocessors

If you look in the `app/assets/stylesheets` folder you'll see several files. The files `home.css.scss` and `recipes.css.scss` were created when we generated the `Home` controller and the `Recipes` controllers. The `scaffolds.css.scss` file was created for us when we used the Scaffold generator. The `scss` extension on these files means that they are Sass stylesheets, not regular stylesheets.

Sass stands for Syntactically Awesome Style Sheets, and that's a pretty appropriate name. Sass gives developers the flexibility they crave, including the ability to do logic, use variables, and even define functions. Sass files aren't usable by the browser, and so we use a preprocessor to convert the Sass code into regular CSS

Rails automatically preprocesses Sass files for us as a part of the Asset Pipeline. As long as we name the files with the extension `.css.scss`, Rails will do all of the work.

Interestingly enough, if you created a file called `home.css.scss.erb`, it would actually run through the ERb processor first, looking for any sections with <%= %> blocks in the file. It would then pass the results on to the Sass preprocessor so that it could be processed. Preprocessors give you incredible flexibility to create stylesheets for your project.

We can mix CSS and Sass files in the `app/assets` folder, but Rails encourages the use of Sass in projects, so let's look at what Sass is all about so we can use it on our site.

A brief tour of Sass

Let's look at some of the most useful features of Sass, starting with variables.

We can define a variable for a certain color, and then use that color everywhere we want without duplicating the color code, like this:

```
$mainColor: #999;
header, footer{
  background-color: $mainColor;
}

.wrapper{
  border: 1px solid $mainColor;
}
```

When this gets converted to CSS, the variables get replaced with the actual values. The result looks like this:

```
header, footer {
  background-color: #999999;
}

.wrapper {
  border: 1px solid #999999;
}
```

We can use variables for other things as well. For example, we could define a variable called `$width` and set its value, but we can the do mathematical operations on that variable for use in other places.

```
$width: 960px;
    .wrapper{
      width: $width;
    }

    .content{
      width: $width * 0.75;
    }

    .content{
      width: $width * 0.25;
    }
```

Sass will transform that and turn it into this:

```
.wrapper{
  width: 960px;
}

.content {
  width: 720px;
}

.content {
  width: 240px;
}
```

Look at that carefully. Sass took the assigned value of `960px`, dropped off the unit of measure, did the subtraction, and then added the unit of measure back. This works for `em`, `%`, and other units as well. Instead of breaking out the calculator, let Sass do the work!

One other amazingly powerful feature of Sass is that it lets us nest selectors. For example, if you wanted to ensure that all of the hyperlinks in your navigation bar were a different color than the rest of your hyperlinks, you'd have to write CSS like this:

```
nav a {
  color: white;
}
```

But with Sass, you can nest the selectors. This would create child selectors:

```
nav{
  a{
    color: #fff
  }
}
```

This Sass code will be transformed into this CSS:

```
nav a {
  color: inherit;
}
```

As you can see, this nesting uses a descendent selector. But if you wanted to use a child selector instead of a parent selector, you'd do it like this:But you can also use the & character which is a shortcut to the “parent” selector. So, we'd use this code:

```
nav{
  &>a{
    color: #fff
  }
}
```

which, when converted by Sass, will generate a descendent selector like this:

```
nav > a {
  color: inherit;
}
```

This nesting really comes in handy when representing something more significant like a navigation bar that uses an unordered list. The HTML structure for something like that might look like this:

```
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
  </ul>
</nav>
```

which we can represent using Sass's nesting features and the & selector:

```
nav{
  background-color: #ddd;
  float: left;
  width: 100%;

  &>ul{
    margin: 0;
    padding: 0;

    &>li{
      margin-right: 1%;
      color: #fff;

      &>a{
        color: inherit;

        &:hover, &:focus{
          color: #000;
        }
      }
    }
  }
}
```

The CSS structure more closely mirrors our HTML. It even makes it easier to do the hover states for the HTML links! And when it gets converted for use in the browser, it'll have the child selectors applied appropriately, like this:

```
nav {
  background-color: #ddd;
  float: left;
  width: 100%;
}

nav > ul {
  margin: 0;
  padding: 0;
}

nav > ul > li {
  margin-right: 1%;
  color: #fff;
}

nav > ul > li > a {
  color: inherit;
}

nav > ul > li > a:hover, nav > ul > li > a:focus {
  color: #000;
}
```

We can put all of this to use inside of a Rails application thanks to Rails' asset pipeline. Let's do that now.

9.2. Creating Some Basic Styles

Right now, Rails is loading up all of the files in the `app/assets` folder. But the files that are currently there need to be cleaned out.

The files `home.css.scss` and `recipes.css.scss` are completely empty. They're generated so that if we wanted, we could have specific styles for specific

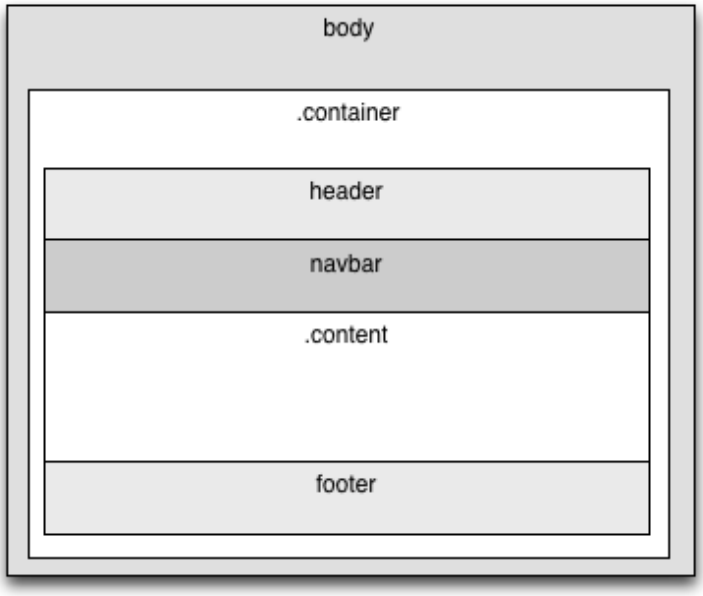
controllers. But let's get rid of those by deleting them from our project. We won't be needing them.

The `scaffolds.css.scss` contains some basic styling for the error messages, the fonts, and a few other areas. Review its contents, but then remove this file from your project as well. We'll construct our own stylesheets.

Defining a Layout

We've used HTML to define the structure of our layout in the file `app/views/layouts/application.html.erb` but now it's time to make that layout look a little nicer. Before doing that, let's look at the layout again:

Figure 9.1. Our Layout



This represents the structure of our HTML, not necessarily how we'd like the page to look. We do that with CSS, and we need to be very mindful of our audience. We probably want to make sure that whatever we build will work well on mobile devices, tablets, and desktops alike. So, we'll be sure to incorporate responsive design into our CSS.

Create a new file called `app/assets/stylesheets/layout.css.scss`. We'll use this file to define the basic layout for our site. We'll start off by creating a few variable declarations at the top of the file:

Example 9.3. code/08_assets/cookbook/app/assets/stylesheets/layout.css.scss

```
$backgroundColor: #ddd;
$contentBackgroundColor: #fff;
$hoverColor: #999;
$textColor: #333;
$width: 960px;
```

Declaring variables like this at the top of the file makes it really easy to locate and change them later. For example, here we're defining the width of the page along with some colors. We'll use these throughout this file, and if we decide to change the width or these colors, we can simply pop open the file, make the change in one spot, and call it good.

Now, we'll define a simple rule for the page's `body` element that defines the background color and the text color:

Example 9.4. code/08_assets/cookbook/app/assets/stylesheets/layout.css.scss

```
body{
  background-color: $backgroundColor;
  color: $textColor;
}
```

Then let's define a rule for the container element, which wraps our page's header, main content, and footer. On desktop screens, we'll constrain the page's width to 960 pixels wide. and on devices with a screen width that's smaller than 960 pixels, we'll make the container scale down.

To do this with regular CSS, we'd need to repeat ourselves a little bit, by defining variations of CSS rules based on media queries. However, Sass allows us to embed media queries *inside* of selectors! So we can define our rule like this:

Example 9.5. code/08_assets/cookbook/app/assets/stylesheets/layout.css.scss

```
.container{
  background-color: $contentBackgroundColor;
  margin: 0 auto;

  @media only screen and (min-width: $width){
    width: $width;
  }

  @media only screen and (max-width: $width -1){
    width: 90%;
  }
}
```

This makes it more clear how the different screen sizes affect the styles. We can see all of the styles associated with this element at once, instead of scrolling up and down the file looking at the various media queries. When the actual CSS gets rendered, the regular media queries get generated for us as if we'd written them ourselves.

Now let's add some styles for the content area, the header, and the footer:

Example 9.6. code/08_assets/cookbook/app/assets/stylesheets/layout.css.scss

```
.content, header, footer{
  padding: 1%;
}

footer{
  text-align: center;
}
```

With those in place, we can turn our attention to the navigation bar. We'll use Sass's nesting feature here again. First we'll define a variable at the top for the navbar's background color:

Example 9.7. code/08_assets/cookbook/app/assets/stylesheets/layout.css.scss

```
$navColor: #ccc;
```

and then we'll add the CSS for the navigation bar and the links within:

Example 9.8. code/08_assets/cookbook/app/assets/stylesheets/layout.css.scss

```
nav{
  background-color: #ccc;
  text-align: center;

  &>a{
    color: inherit;
    padding: 1%;
    text-decoration: none;

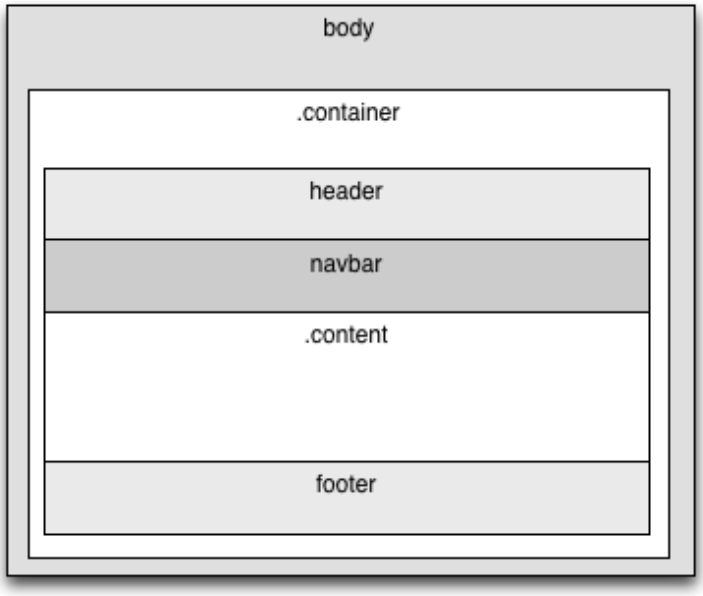
    &:hover, &:focus{
      color: $hoverColor;
    }

  }
}
```

We use the `&` syntax for the `:hover` and `:focus` pseudoclass definitions here as well.

If you open this app in the browser you'll see something that looks like this:

Figure 9.2. Our design so far



And it should look great scaled down on a smaller display, too, thanks to the media queries we embedded. That takes care of the main layout. Now let's turn our attention to the forms and error messages.

9.3. Styling Forms and Errors

When we built our form, each label and field is wrapped inside of a `div` element:

```
<div class="field">
  <%= f.label :title %><br>
  <%= f.text_field :title %>
</div>
```

And so is the form's Submit button:

```
  <div class="actions">
    <%= f.submit %>
  </div>

<% end %>
```

Let's space out the form fields a little bit by adding some bottom margin to these `div` elements:

Example 9.9. code/08_assets/cookbook/app/assets/stylesheets/forms.css.scss

```
div.field, div.actions {
  margin-bottom: 10px;
}
```

Next, let's look at how errors are displayed. We'll need to give some visual feedback to our users when they make a mistake filling out our forms. We had some ugly (and hard to read) feedback that we got for free from the Rails scaffolding, but we removed that when we removed the `scaffolds.css.scss` file, so we'll need to add some styles back in.

The Rails helpers we used for our form fields will automatically wrap the labels and form fields with `div` elements with the class of `field_with_errors` like this:

```
 <div class="field">
  <div class="field_with_errors"><label for="recipe_title">Title</label></div><br>
  <div class="field_with_errors"><input id="recipe_title" name="recipe[title]"
 type="text" value="" /></div>
</div>
```

This lets us easily highlight the error fields. So to do that let's create a variable we can use for the highlighting color. Create a new Sass stylesheet file called `app/assets/stylesheets/forms.css.scss` and add this variable declaration to the top:

Example 9.10. code/08_assets/cookbook/app/assets/stylesheets/forms.css.scss

```
$errorColor: #c00;
```

Then add this declaration to style the form fields when there are errors:

Example 9.11. code/08_assets/cookbook/app/assets/stylesheets/forms.css.scss

```
.field_with_errors {
  background-color: $errorColor;
  color: #fff;
  display: inline-block;
  padding: 1px;
}
```

Our current form also shows an error explanation region at the top of the page when there are errors. This additional explanation is great for giving the user a summary of what they need to fix. But let's add a tiny bit of styling for this region as well:

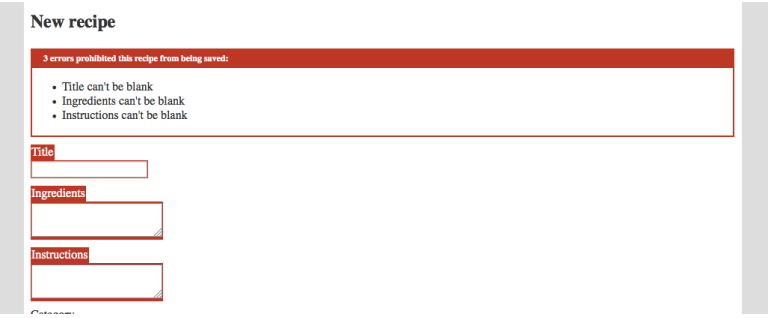
Example 9.12. code/08_assets/cookbook/app/assets/stylesheets/forms.css.scss

```
#error_explanation {
  border: 2px solid $errorColor;
  margin-bottom: 10px;

  h2 {
    background-color: $errorColor;
    color: #fff;
    font-size: 12px;
    margin: 0;
    padding: 5px 5px 5px 15px;
  }
}
```

And with that, our error page looks like this:

Figure 9.3. Our error page



9.4. What You Learned

In this chapter you learned how to use Rails' asset pipeline and Sass to build easy-to-manage styles for your application. You learned how to leverage variables in Sass and how to split your styles into separate files for easy maintainability.

Exercises

1. Create a new file called `app/assets/notice.css.scss` and create some style rules to style the “notice” region of the page. This is the region that displays the success message when you create, update, or delete records.

- Use variables for any colors you decide to use
- Test your notice message on both small and large screen displays.

2. In `app/assets/stylesheets/forms.css.scss`, style the text field and text areas so they are taller and wider. Make sure that any sizing you do works on both large and small screens.

At this point we have enough of an application built that we can share it with the world, so let's look at how we can put this application into production!

Chapter 10. Using Git and Deploying To Heroku

We've built an app, and now it's time to show it off publicly. But to do that we'd better start thinking more about how we'll manage that code, get it on a production server, and then manage changes so we can push new versions of our application to production easily. There are many options for deploying Rails applications, but Heroku is one of the easiest solutions available.

Heroku is a cloud-based hosting provider that makes it easy for developers to share their apps with the rest of the world. They provide a free plan, too, so if you're just getting started with your idea, you don't have to cough up money or a hosting plan.

We'll spend this chapter walking you through setting up an account with Heroku. Along the way, we'll create a Git repository for your Rails application so you can learn to track changes to your project and make updating your live application a breeze.

10.1. Preparing For Launch

To use Heroku effectively, you'll need to have the Heroku Toolbelt installed. Heroku also requires you to use the Git version control system for your application so we'll learn how to set that up and work with Git repositories. You'll also need to have an SSH keypair created so you can push your application to Heroku over SSH, but we'll work through creating that keypair if you don't have one already. And of course, you'll need an account at Heroku to make all of this happen.

Installing the Heroku toolbelt

Heroku provides a commandline utility you can use to manage and create applications easily. Install this by visiting https://toolbelt.heroku.com/ and choosing the version for your operating syste,

Adding Required Gems

Heroku does make it easy to deploy a Rails application without configuring a database or a web server on a Linux environment, but we have to follow some very specific steps to make things work in its environment.

First, Heroku doesn't use SQLite as its database, and that's for the best since SQLite isn't designed for use by hundreds or thousands of end users. Heroku uses the free and incredibly powerful Postgres database. In order to use that, we must modify our `Gemfile` to include support for this database. We also need another gem called `rails_12factor` to make Rails logging and asset handling work properly with Heroku. We can actually configure Gems for specific environments, and so we'll specify that these two new gems should only exist in the `production` environment:

Example 10.1. code/09_latest_recipe/cookbook/Gemfile

```
group :production do
  gem 'pg'
  gem 'rails_12factor'
end
```

Now, any gems listed in our Gemfile that don't specify the environment will be installed in all environments. We don't need the SQLite gem on our production Heroku environment, so we'll modify the the Gemfile to only use the SQLite3 gem in the `development` and `test` environments:

Example 10.2. code/09_latest_recipe/cookbook/Gemfile

```
# Use sqlite3 as the database for Active Record
gem 'sqlite3', group: [:development, :test]
```

Whenever we change to the Gemfile, we need to run the `bundle install` command. However, we want to use a new option that skips the installation of production stuff. You may not have all of the prerequisites you need on your computer to correctly install the `pg` or `rails_12factor` gems that Heroku needs. So we'll tell Bundler to ignore the `production` gems when we run this on our machine:

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

Creating a Heroku account

Visit http://heroku.com/ and sign up for an account using a valid email address. When you provide your email address they'll send you an activation email. When you receive that email, follow the activation link and complete the registration process by providing a password.

Once you have an active account, you need to associate your public SSH keys with your account.

Logging In to Heroku

Run the command

```
heroku login
```

This will prompt Heroku to ask us for our Heroku account credentials. It will also attempt to associate your SSH keys with Heroku. But don't worry - if you don't have an SSH key, the `login` utility will create one for you!

```
heroku login
Enter your Heroku credentials.
Email: brian@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/brian/.ssh/id_rsa.pub
```

If you're on Windows and you've used the RailsInstaller program, you probably already have an SSH key that Heroku will find and associate with your account.

10.2. Managing Source Code With Git

In order to upload our application to Heroku so it can be deployed, we need to use the Git version control system. let's briefly learn about Git and how it works.

Git In A Nutshell

Git is a version control system. It lets you track all of the changes that you make to your code. Instead of having to keep multiple backup copies of each file, or keep large chunks of code commented out “just in case”, you make periodic snapshots of your source code, called “commits”. You can then review the logs of your commits, which lets you and your teammates see who's responsible for each line of code in the project.

Git is great for solo developers as well as large teams. You can share your Git repository with others over the network or through services like Github [1] or Bitbucket [2] or something you set up yourself using SSH. You “push” a copy of your repository to a remote repository, and Git merges in the changes.

Not only does Heroku offer application hosting, they also offer Git repository hosting as well. When you push your code to Heroku, it automatically deploys your application. So that means we'd better start using Git to manage our code!

Creating A New Git Repository

In the root of your Rails application, type:

```
git init
```

```
Initialized empty Git repository in /Users/bphogan/cookbook/.git/
```

This creates a blank repository, which means it doesn't know anything about our files. We need to fix that by using Git to add our files to the repository. We can do that easily with

```
git add .
```

This command tells Git to add every file and folder in our application to its list of files to watch for changes. We're still not done though.

Once we've told Git about our files, we need to “commit”, or tell Git that it should take a snapshot of our directory structure at this point in time. This snapshot is what

[1] http://github.com
[2] http://bitbucket.com

gets deployed to Heroku. it also gives us a point in time to go back to if we make mistakes later.

To create the snapshot, type this command:

```
$ git commit -m "Ready for deployment"
```

The `-m` flag lets us provide a commit message, and this message is really important when we go back a few months later to figure out what we did. You must supply a commit message whenever you make a commit.

10.3. Deploying the Application

With our Git repository ready, we can create a new application on Heroku like this:

```
$ heroku create bphrails3cookbook
```

Names must be unique!

Application names on Heroku have to be unique, so I recommend using your initials or something specific to your app's name.

You should see this message, which confirms your app is created and linked to your repository:

```
  Creating bphrails3cookbook..... done
Created http://bphrails3cookbook.heroku.com/ | git@heroku.com:bphrails3cookbook.git
Git remote heroku added
```

You can now push your application to the cloud with this command:

```
$ git push heroku master
```

This pushes your master branch to a remote Git repository on Heroku.

```
Counting objects: 102, done.
          Delta compression using up to 2 threads.
          Compressing objects: 100% (86/86), done.
          Writing objects: 100% (102/102), 93.32 KiB, done.
          Total 102 (delta 3), reused 0 (delta 0)

          -----> Heroku receiving push
          -----> Removing .DS_Store files
          -----> Rails app detected
          -----> Detected Rails is not set to serve static_assets
                 Installing rails3_serve_static_assets... done
          -----> Gemfile detected, running Bundler version 1.0.0
                 Unresolved dependencies detected; Installing...
                 Fetching source index for http://rubygems.org/
                 ...
                 Your bundle is complete! Use `bundle show [gemname]` to see where
 a bundled gem is installed.

                 Your bundle was installed to `.bundle/gems`
                 Compiled slug size is 3.9MB
          -----> Launching.... done
                 http://bphrails3cookbook.heroku.com deployed to Heroku
```

Once Heroku received your code, it ran some scripts to copy your files to a place where it could make them publicly available on the web.

Setting up your Database

We've deployed our application with Git, but this process doesn't set up our database. However, we can use some Rake tasks to do that, using Heroku's command line interface. From the console, run this command:

```
$ heroku run rake db:setup
```

This runs the `db:setup` Rake task on our Heroku slice. This creates the tables in our database and then runs the contents of the `db/seeds.rb` file.

We can use this to run any Rake task you need to run, including migrations later on.

Restarting the Application

In production mode, our Rails application doesn't notice changes to our database or our files. We've changed the database, so we need to restart the server to make our application start working. Type this command in the console to restart your application:

```
$ heroku restart
```

Once that's done, you can view your application at yourappname.heroku.com and share it with the world!

10.4. Building and Deploying a New Feature With Git

Let's make a minor change to our application's home page - let's display the most recent recipe if one exists. We'll use Git to track our changes and push this new feature to production.

First, let's make a new branch in our repository. A branch lets us do work in a separate part of our codebase, isolated from the rest of the code. If we mess up, or decide we don't want the feature, we can throw away the branch and return to our main repository.

Git defaults to the “master” branch, but it's considered a bad practice to do any work on the master branch. Instead, you should always create a new branch to do your work in. So let's do that:

```
$ git branch latest_recipe
```

Once we've created the new branch, we need to "check it out" so we can work on it.

```
$ git checkout latest_recipe
```

Now we can work on this new bit of functionality

Branch and Checkout

We could have also used the command `git checkout -b latest_recipe` to create and check out a new branch.

Adding the Latest Recipe

To get the latest recipe from the homepage, we need to get the latest recipe in the controller for the home page. We can do that by using the `last` method on the `Recipe` class. Open up `app/controllers/home_controller.rb` and change the `index` action so it looks like this:

Example 10.3. code/09_latest_recipe/cookbook/app/controllers/home_controller.rb

```
def index
  @number_of_recipes = Recipe.count
  @latest_recipe = Recipe.last
end
```

Then we just add this code to `app/views/home/index.html.erb`:

Example 10.4. code/09_latest_recipe/cookbook/app/views/home/index.html.erb

```
<h3>Latest Recipe</h3>
<% if @latest_recipe %>
  <h4><%=@latest_recipe.title %></h4>
  <p><%=@latest_recipe.ingredients %></p>
  <p><%=@latest_recipe.instructions %></p>
<% else %>
  <p>There are no recipes to display</p>
<% end %>
```

Here we're checking to see if there's a latest recipe. If there is, we display that recipe, and if we don't have one, we just tell the user there aren't any.

Committing and Merging Your Branch

When you have a working feature, you need to integrate that feature into the rest of your code. To do that, you need to tell Git to take another snapshot of your code. We do that the same way we did before-by telling Git to track some files and then take the snapshot.

```
git add .
```

```
git commit -m "added latest recipe"
```

Now all we have to do is merge our new snapshot into our old snapshot. We do that by checking out our "master" branch, which is our main snapshot.

```
git checkout master
```

Then we merge our `latest_recipe` branch *into* our `master` branch:

```
git merge latest_recipe
```

With that, our `master` branch is updated. We can optionally throw away our feature brach with `git branch -d latest_recipe`.

Deploying Our Changes

We deploy the changes to Heroku in the same way we did before - by pushing our changes to the `heroku` remote Git repository.

```
$ git push heroku master
```

This deploys and restarts the application. Now you can go and share this with your friends and family!

10.5. What You Learned

We've covered two important things in this chapter. First, you learned how to put your app into production on Heroku using their free teir. Second, you learned how to manage your project's source code using the Git version control tool.

Exercises

1. When a category is deleted, set all associated recipes to a nil category. Look at the `has_many` relationship on the `Category` model and then look at the documentation for ActiveRecord associations to see if you can find the right option.

    - Create a new branch in Git
    - Make the change to the association
    - Ensure the change works as expected
    - Commit the changes to your ranch
    - Switch to the master branch and merge the changes from your branch to your master branch
    - Deploy your changes to Heroku.

Next, let's look at a few more features of the Rails framework that you may need to use in the future.

Chapter 11. Beyond The Basics

There's a lot more to the Rails framework than what we covered here. Let's explore a few other features.

11.1. Writing Documentation with RDoc

Documenting code is one of the most useful things a developer can do. Unfortunately it's often done poorly if it's even done at all.

Ruby on Rails aims to change how developers write documentation by making use of RDoc. RDoc is a program that can parse Ruby files for comments and convert these comments to HTML pages or other formats. It generates very clean and nice-looking documentation and is so easy to use that developers quickly come to actually enjoying documentation.

Any comments located directly above a class or method declaration will be interpreted by the RDOC parser to be the comments for that given block of code.

Here's an example of some commented code.

```
 1 #=Recipes
 2 # Recipes are added, removed, maintained, and viewed using
 3 # the actions in this controller.
 4 #==Authentication
 5 # There is no authentication on this controller
 6 class RecipesController < ApplicationController
 7
 8   # This action handles the default document (Index) and
 9   # simply redirects users to the list action.
10   def index
11     list
12     render :action => 'list'
13   end
14
15 end
```

When we run the command `rake doc:app`, our HTML documentation will be created for us. See Figure 11.1, "RDoc output in HTML" on page 115

Figure 11.1. RDoc output in HTML

11.2. Working with the Console

Sometimes, when working with Rails, you might want to try something out that you don't quite understand. Other times you want to explore the objects in your application, or even just quickly create some records. As you saw earlier, the Rails Console lets you load the entire Rails application into the Interactive Ruby environment so you can execute commands and interact with your application. This can be great for debugging, or even manipulating records in an application.

From the root of your project, execute the command

```
rails console
```

to enter the console. Once the console is loaded, you can start experimenting with your objects, as shown in Figure 11.2, “Using the Rails Console to work with objects” on page 117

Figure 11.2. Using the Rails Console to work with objects

```
Loading development environment (Rails 4.0.1)
:001 > recipe = Recipe.create :title => "test", :ingredients =>
 "stuff", :instructions => "mix together"
  D, [2013-11-17T15:02:56.306420 #64577] DEBUG -- :    (0.1ms)  begin transaction
  D, [2013-11-17T15:02:56.326046 #64577] DEBUG -- :   SQL (9.8ms)  INSERT INTO
 "recipes" ("created_at", "ingredients", "instructions", "title", "updated_at")
 VALUES (?, ?, ?, ?, ?)  [["created_at", Sun, 17 Nov 2013 21:02:56 UTC +00:00],
 ["ingredients", "stuff"], ["instructions", "mix together"], ["title", "test"],
 ["updated_at", Sun, 17 Nov 2013 21:02:56 UTC +00:00]]
  D, [2013-11-17T15:02:56.328395 #64577] DEBUG -- :    (1.0ms)  commit transaction
   => #<Recipe id: 2, title: "test", ingredients: "stuff", instructions: "mix
 together", created_at: "2013-11-17 21:02:56", updated_at: "2013-11-17 21:02:56",
 category_id: nil>
:002 > id = recipe.id
  => 2
:003 > recipe.category
  => nil
:004 > category = Category.find_by_name "Sandwiches"
  D, [2013-11-17T15:06:03.150904 #64577] DEBUG -- :   Category Load (0.7ms)
 SELECT "categories".* FROM "categories" WHERE "categories"."name" = 'Sandwiches'
 LIMIT 1
  => #<Category id: 6, name: "Sandwiches", created_at:  "2013-11-17 19:22:40",
 updated_at: "2013-11-17 19:22:40">
:005 > recipe.category = category
:006 > recipe.changed
  => ["category_id"]
:007 > recipe.save
  => true
:008 > exit
```

Then I use the find method on Recipe to locate the recipe again. Then I see if it has a category. Of course, it doesn't so I fetch a category from the database and assign it to my instance. I use the `changed` method to see what columns in the database have changed. The association is not saved until I execute the save method of my instance.

This is just the beginning, but it shows how you can use the console to learn more about how the methods on the classes work without having to write any view pages or controller code. For example, on a relatively small application, you could use the Console to create and modify data instead of spending the time building a user interface.

11.3. Logging

Rails applications automatically log requests and responses to the various logs. One log you should really keep an eye on is your development.log file. It contains a lot of useful information such as the parameters sent on each request as well as the SQL statements created by Rails and sent to the database.

This is the place you'll want to look to tune your application. Not only can you see if you're executing too many SQL statements for the job at hand, but you can also see how long it took Rails to serve the request to your client.

11.4. Writing your own SQL statements

At first glance, Rails may seem limited. We've gone through this entire project without writing any SQL. A lot of the time we won't have to worry about it. However, it is still very possible for us to get into the code and do what we need to do.

For example, one of the methods in Active Record is called `find_by_sql` which allows us to look records up using our own custom SQL statement.

```
@results = Recipe.find_by_sql "select r.title, c.name
  from recipes r
  join categories c
  on r.category_id = c.id"
```

```
  => [#<Recipe:0x3750478 @attributes={"name"=>"Beverages", "title"=>"Test"}>]
```

You have to understand Ruby to understand what this example returns, so I'll help out. The square brackets ([]) surrounding the result means that you're dealing with an `Array`. The `#<Recipe` piece means it's a Recipe object. So when you use the `find_by_sql` method, you receive an array of objects which you can then iterate over.

```
@results.each do |recipe|
  puts recipe.name  # print to STDOUT
  puts recipe.title # print to STDOUT
end
```

Note that a new method `name` has been created in the instance of the `Recipe` object. Active Record inspected the column names that came back from the database and dynamically created accessor methods for us to use.

Warning

Never use "puts" in your Rails application directly. It can cause problems that you may not find later on. It's only to be used in tests and in the console to help you debug. If you're wondering, it's equivalent to `system.out.println` in Java, `echo` in Bash and PERL, or `console.log` in JavaScript. It pushes its output to STDOUT in the console.

There are many other features in Rails that make it extremely flexible. Don't get fooled into thinking Rails is all about scaffolding. Rails is much more than that!

11.5. What You Learned

In this chapter you learned how to create documentation with RDoc which makes it easier for developers to understand how your application works. You also learned how Rails logging works, and you explored how to write your own SQL statements when the ones that Rails creates for you aren't going to be efficient enough. From here you've got the solid foundation you need to continue exploring the Rails framework.

Exercises

1. Document some other methods in your controllers and models and then regenerate the docs. Here are some simple formatting symbols:

   - # is the comment
   - = is a large header
   - == is a level 2 header
   - --- is a horizontal rule
   - * is a bullet

That does it for the tutorial, but in the next chapter you'll find some additional resources you can use to continue your exploration of the Rails framework and the Ruby programming language.

Chapter 12. Where To Go Next?

Hopefully this small tutorial gave you enough of an overview of the Ruby on Rails framework to see the potential impact it has on rapid application development. From here, you should be able to extend this assignment by doing the homework or explore further by coming up with your own application, using this as a model.

12.1. More On Deploying Rails Applications

Heroku makes deploying Rails applications extremely simple, but Heroku does have limitations. It's a read-only file system, so you need to use other cloud services like Amazon S3 if you want to upload files to your application. If you need something more complex, you'll need to look at other options.

Rails applications require more setup work than PHP applications and there are lots of things that can go wrong when you're first starting out. Do not attempt deployment until you are very comfortable with the Rails framework.

Applications can be deployed using a variety of methods, but the most popular method is to use Passenger. Passenger works with Apache or Nginx to make deploying Rails applications extremely easy. The Passenger site[1] provides more information on deploying applications using this method. Another excellent method for deployment is Apache or Nginx balancing a collection of Thin server instances; however this method is more difficult to manage than Passenger.

There are many web hosting companies that support Rails. Shared hosts such as Dreamhost keep the cost low by sharing space and memory with other users. This is a great low-cost way to launch your application. Dreamhost makes hosting Rails applications painless by employing Passenger, an Apache module designed to make deploying Rails applications easy.

If you need high availability, you can look at EngineYard [http://www.engineyard.com/], which promise to host large Rails sites with ease. Engine Yard is significantly more expensive than shared hosting plans. You can't afford it unless you are really making money. It's worth every penny though and they do offer a Solo plan using Amazon's EC2 cloud.

If you just want to set things up yourself on dedicated virtual servers, you could look Linode [http://www.linode.com/], and DigitalOcean [http://www.digitalocean.com/]. You'll set up your deployments using Git and Capistrano.

Deploying applications is not trivial, but it's not a terrible experience either. Once you've gotten your head around how it all works, you'll have no problem moving apps around.

[1] http://www.modrails.com/

More information on deployment can be found in the book *Deploying Rails Applications*[2] from the Pragmatic Bookshelf.

12.2. Development Tools

There are several different tools you can use to easily build Rails applications.

- First, check out SublimeText. While not an IDE, it is built for developers and has great support for Rails. It's also available for Windows, OSX, and Linux. See http://http://www.sublimetext.com/ for more information.

- Those interested in a full-blown IDE for Rails development will love RubyMine from JetBrains. See http://www.jetbrains.com/ruby/. Rubymine is a fantastic editor with excellent Rails and Ruby support for large projects.

- If you're comfortable using VIM, you want to look into the rails.vim plugin.

  Rails.vim plugin: http://www.vim.org/scripts/script.php?script_id=1567

12.3. Books

- Learn to Program (Chris Pine)

  http://www.pragprog.com/titles/fr_ltp/learn-to-program

- Agile Web Development with Rails, Fourth Edition (Sam Ruby, Dave Thomas, et all)

  http://pragprog.com/book/rails4/agile-web-development-with-rails-4

- Pragmatic Guide to Sass (Hampton Catlin and Michael Lintorn Catlin)

  http://pragprog.com/book/pg_sass/pragmatic-guide-to-sass

- Pragmatic Guide to Git (Travis Swicegood)

  http://pragprog.com/book/pg_git/pragmatic-guide-to-git

- Programming Ruby (Dave Thomas)

  http://pragprog.com/book/ruby4/programming-ruby-1-9-2-0

- The Well Grounded Rubyist (David A. Black)

  http://www.manning.com/black2/

[2] `http://pragprog.com/titles/cbdepra`

- HTML5 and CSS3 (Brian P. Hogan)

  http://pragprog.com/book/bhh52e/html5-and-css3

- Practical Object-Oriented Design in Ruby (Sandi Metz)

  http://www.poodr.com/

- Deploying Rails Applications (Anthony Burns and Tom Copeland)

  http://pragprog.com/book/cbdepra/deploying-rails

12.4. Online Resources

- Try Ruby (http://tryruby.org/)
- Railscasts (http://railscasts.com/)
- The Rails Tutorial (http://ruby.railstutorial.org/)
- #rubyonrails IRC channel (http://wiki.rubyonrails.org/rails/pages/IRC)
- Rails Mentors (http://railsmentors.org/)
- Git Immersion (http://gitimmersion.com/)

Index

A

accessor, 53
Altering database tables, 77
Array, 118
Arrays, 45
  mapping records with collect, 80
  mapping records with map, 80
Asset Management, 5
Associations
  belongs_to, 78
  eager loading, 81
attr_accessor, 43
attr_reader, 43
attr_writer, 43

B

basic authentication, 89
  limitations, 91
  securing actions with, 90
before filters
  authenticating users with, 89
  refactoring code with, 85
  sharing code with, 59
Blocks, 46
  defining DSLs, 47
  iterating with, 46
Bundler
  grouping gems, 107
  installing gems with, 14
  managing dependencies with with, 14
  specifying targets, 108
button_to, 62

C

class, 53
Classes, 41
classes
  defining, 42
Comparison operators, 40
Console
  modifying records with, 116
control flow, 40
Controllers, 2, 27

D

data
  populating database with seed data, 76
database.yml, 28
deleting records, 62
Deploying, 121
deployment
  using Heroku, 107
development environments, 122
Documentation
  generating documentation with RDOC, 115
  using RDoc, 115
Domain Specific Language, 47
DSL, 47

E

error messages
  styling, 103
Exception Handling
  rescue, 84

F

Fixtures, 70
flash session
  providing feedback with, 65
Foreign Keys
  Rails support foreign, 78
form helpers
  select, 80
functions
  alias for Methods, 41

G

Gems, 5
  Heroku-specific gems, 107
  managing with Bundler, 14
Generators, 25
  new project, 12
  scaffold, 26
Git, 109
  adding files to a repository, 109
  Branching, 111
  branching, 112
  checking out branches, 112
  commit, 109

creating a repository, 109
deploying with, 109
merging a branch, 112
repository, 109
sharing code with, 109
staging files for committing, 109
switching between branches, 112
switching branches branches, 112

H

Hashes, 45
helper methods, 63
Heroku
git repositories, 109
HTML5, 18

I

if, 40
Installing Ruby
on Microsoft Windows, 6
on OSX, 7
on Ubuntu, 8
instance variables, 43, 64
Integration testing, 5

J

JavaScript
deleting records, 62

L

Layouts, 18
LEFT JOIN
generating for single records, 83
performance issues, 83
link_to, 62
mapping links to controllers and actions, 19
Links
linking resources, 54
logging, 117
SQL statements, 117
Looping
iterating with methods, 47

M

media queries, 100
Methods, 41
methods
class methods, 44
defining, 41
Migrations, 5, 30
adding columns to tables with, 77
adding foreign keys with, 77
running, 31
Model Tests, 69
Model-View-Controller pattern, 2, 2
Models, 2
ActiveRecord, 52
dynamic method creation, 52
inner workings of, 52
MVC, 2

N

navigation, 19
notice, 65
Numbers, 37

O

Objects, 41

P

Partials
sharing code with, 56
Preprocessors, 96
puts, 118

R

Rails
installing multiple versions of, 9
using specific versions, 13
Rake, 6
loading data with, 76
Rake tasks
Cloning the test database from the development database, 71
RDoc, 115
example, 115
Generating documentation, 115
reflection, 53
Relationships, 77
responsive design, 100
REST, 62

Restarting the web server, 78
Routes, 20
  form routing, 60
  named routes, 20
  root route, 20
Routing, 20
  Default routes, 20
routing helper methods, 65
Rubygems
  Gems, 6
RVM, 7
  installing on OSX, 7
  installing on Ubuntu, 8

S

Sass, 96
  computing values, 97
  nested selectors, 97
  using media queries, 101
  variables, 96, 101
Scaffolding, 25
  issues with, 28
  limitations of, 25
  use as a learning tool, 25
  use in production, 25
Security, 80
security
  hiding elements, 91
server, 31
Sessions, 90
sessions
  cookie store, 90
SQL
  statements in the Rails logs, 117
  writing your own, 117
Static methods, 44
String interpolation, 39
Strings, 37
Strong parameters, 80
Symbols, 45, 67

T

Test-driven development, 5
Tests
  running a single test, 71
  running all unit tests, 71
time_ago_in_words, 63

U

Unit testing, 5
Unit Tests, 68, 69
  running, 71
unless statements, 41

V

Validations, 67
  validates_presence_of, 67
Variables, 37
Views, 2
  displaying errors, 72
  master layout, 18
views, 17
  customizing generated views, 17
  instance variables in, 64

W

web server
  using WEBRick, 16
WEBrick, 16