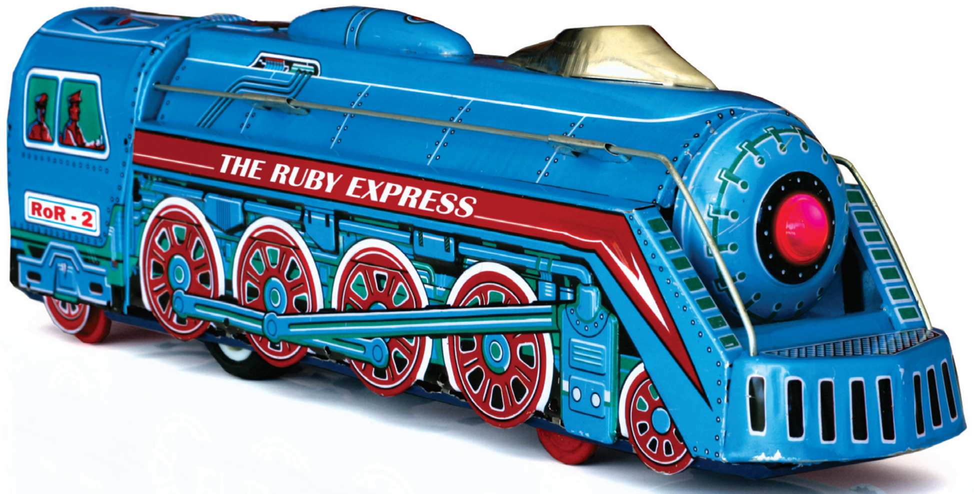
sitepoint®

SIMPLY RAILS 2

BY PATRICK LENZ



THE ULTIMATE BEGINNER'S GUIDE TO RUBY ON RAILS

Simply Rails 2
(Chapters 1, 2, 3, and 4)

Summary of Contents of this Excerpt

Summary of Additional Book Contents

sitepoint®

SIMPLY RAILS 2

BY **PATRICK LENZ**

SECOND EDITION

Simply Rails 2

by Patrick Lenz

Expert Reviewer: Luke Redpath
Managing Editor: Simon Mackie
Technical Editor: Andrew Tetlaw
Technical Director: Kevin Yank
Printing History:
First Edition: January 2007
Second Edition: May 2008

Editor: Hilary Reynolds
Index Editor: Max McMaster
Cover Design: Alex Walker

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9804552-0-5
Printed and bound in the United States of America

About the Author

Patrick Lenz has been developing web applications for more than ten years. Founder and lead developer of the freshmeat.net software portal, he and his Rails consultancy and web application development company, limited overload, are responsible for several community-driven web applications developed using Ruby on Rails. Patrick also authored some of the first articles to appear on the web about architecting and scaling larger Rails applications.

Patrick lives in Wiesbaden, Germany, with his wife Alice and his daughter Gwendolyn.

When not working in front of a computer, he can often be seen with a camera in his hand, either taking artsy pictures or documenting the progress of his baby girl conquering the world.[1] He also enjoys cars, music, and extended weekend brunches with friends.

His weblog can be found at http://poocs.net/.

About the Expert Reviewer

Luke Redpath is a programmer with over seven years' experience in the web design and development field. A recovering PHP and ASP developer, Luke has been using Ruby and Rails professionally for nearly two years and has released and contributed to several Ruby libraries and Rails plugins, including UJS—the Rails unobtrusive JavaScript plugin.[2] He currently resides in North London with his long-term partner Julie.

About the Technical Editor

Andrew Tetlaw has been tinkering with web sites as a web developer since 1997 and has also worked as a high school English teacher, an English teacher in Japan, a window cleaner, a car washer, a kitchen hand, and a furniture salesman. At SitePoint he is dedicated to making the world a better place through the technical editing of SitePoint books and kits. He is also a busy father of five, enjoys coffee, and often neglects his blog at http://tetlaw.id.au/.

About the Technical Director

As Technical Director for SitePoint, Kevin Yank oversees all of its technical publications—books, articles, newsletters, and blogs. He has written over 50 articles for SitePoint, but is best known for his book, *Build Your Own Database Driven Website Using PHP &*

[1] His pictures are regularly published to Flickr and are available at http://flickr.com/photos/scoop/
[2] http://www.ujs4rails.com/

MySQL.[3] Kevin lives in Melbourne, Australia, and enjoys performing improvised comedy theater and flying light aircraft.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our books, newsletters, articles, and community forums.

[3] http://www.sitepoint.com/books/phpmysql1/

To my daughter Gwendolyn and
my wife Alice.

Table of Contents

Preface

Ruby on Rails has shaken up the web development industry in a huge way—especially when you consider that version 1.0 of Rails was only released in December 2005. The huge waves of enthusiasm for the new framework, originally in weblogs and later in the more traditional media, are probably the reason why this book is in your hands.

This book will lead you through the components that make up the Rails framework by building a clone of the popular story-sharing web site digg.com. This will give you a chance to get your feet wet building a simple, yet comprehensive web application using Ruby on Rails.

While the first edition of this book hit the shelves shortly after Rails 1.2 was released, the Rails Core Team quickly hurried off to work on an even better and even more polished version of the framework—a version that was released in December of 2007 as Rails 2.0. Although seen as an evolutionary (rather than a revolutionary) update, Rails 2 features improvements in almost every corner of its comprehensive code base, hence the requirement to update this book. And the improvements continue: as we go to press, the 2.1 release of Rails is imminent.

Without going into too many boring details, rest assured that with Rails 2 you have the fastest and most secure, concise, fun and rewarding version of Rails in existence. You get a secure web application almost out of the box; using the latest web technologies such as Ajax has never been more accessible; and it's just as easy to produce a well-tested application as it is not to do any automated testing.

If that's all Klingon to you, don't worry. I'll get you started, and by the time you finish this book, you'll be able to discuss all things Web 2.0 with your friends and coworkers, and impress your dentist with geeky vocabulary.

Who Should Read This Book?

This book is intended for anyone who's eager to learn more about Ruby on Rails in a practical sense. It's equally well suited to design-oriented people looking to build web applications as it is to people who are unhappy with the range of programming languages or frameworks they're using, and are looking for alternatives that bring the fun back into programming.

I don't expect you to be an expert programmer; this isn't a pro-level book. It's written specifically for beginning to intermediate web developers who, though they're familiar with HTML and CSS, aren't necessarily fond of—or experienced with—any server-side technologies such as PHP or Perl.

As we go along, you'll gain an understanding of the components that make up the Ruby on Rails framework, learn the basics of the Ruby programming language, and come to grips with the tools recommended for use in Ruby on Rails development. All these topics are covered within the context of building a robust application which addresses real-world problems.

In terms of software installation, I'll cover the installation basics of Ruby and Ruby on Rails on Mac OS X, Windows, and Linux. All you need to have preinstalled on your system are your favorite text editor and a web browser.

What You'll Learn

Web development has never been easier, or as much fun as it is using Ruby on Rails. In this book, you'll learn to make use of the latest Web 2.0 techniques, RESTful development patterns, and the concise Ruby programming language, to build interactive, database driven web sites that are a pleasure to build, use, and maintain.

Also, as web sites tend to evolve over time, I'll teach you how to make sure you don't wreak havoc with a careless change to your application code. We'll implement automated testing facilities and learn how to debug problems that arise within your application.

What's in This Book?

Chapter 1: Introducing Ruby on Rails

This chapter touches on the history of the Rails framework, which—believe it or not—is actually rather interesting! I'll explain some of the key concepts behind Rails and shed some light on the features that we're planning to build into our example application.

Chapter 2: Getting Started

Here's where the real action starts! In this chapter, I'll walk you through the installation of the various pieces of software required to turn your Mac or PC into a powerful Ruby on Rails development machine. I'll also show you how to set up the database for our example application, so that you can start your application for the first time, in all its naked glory.

Chapter 3: Introducing Ruby
Ruby on Rails is built on the object oriented programming language Ruby, so it helps to know a bit about both object oriented programming and the Ruby syntax. This chapter will give you a solid grounding in both—and if you'd like to get your hands dirty, you can play along at home using the interactive Ruby console.

Chapter 4: Rails Revealed
In this chapter, we start to peel back the layers of the Rails framework. I'll talk about the separation of environments in each of the application's life cycles, and introduce you to the model-view-controller architecture that forms the basis of a Rails application's organization.

Chapter 5: Models, Views, and Controllers
In this chapter, we'll generate our first few lines of code. We'll create a class for storing data, a view for displaying the data, and a controller to handle the interaction between the two.

Chapter 6: Helpers, Forms, and Layouts
This chapter starts off by looking at how Rails's built-in helpers can reduce the amount of code required to create functionality for your application. I'll show you how to use one of the helpers to create a fully functioning form, and we'll style the end result with some CSS so that it looks good! I'll then show you how to write unit and functional tests to verify that the application is working as expected.

Chapter 7: Ajax and Web 2.0
Let's face it, this chapter is the reason you bought this book! Well, it won't disappoint. I'll walk you through the steps involved in adding to our app some nifty effects that use Ajax to update parts of a page without reloading the entire page. Along the way, I'll explain the different relationships that you can establish between your objects, and we'll make sure that our application uses clean URLs.

Chapter 8: Protective Measures
In this chapter, I'll show you how to keep out the bad guys by adding simple user authentication to our application. We'll cover sessions and cookies, and

we'll see firsthand how database migrations allow for the iterative evolution of a database schema.

Chapter 9: Advanced Topics

This chapter will give our example application a chance to shine. We'll add a stack of functionality, and in the process, we'll learn about model callbacks and join models.

Chapter 10: Plugins

In this chapter, I'll show you how to add a plugin—a component that provides features that expand the functionality of your application—to the example application. We'll also talk about some of the more advanced associations that are available to your models.

Chapter 11: Debugging, Testing, and Benchmarking

This chapter will cover testing and benchmarking, as well as the reasons why you should complete comprehensive testing of all your code. We'll also walk through a couple of examples that show how to debug your application when something goes wrong.

Chapter 12: Deployment

Now that you've developed a feature-packed, fully functional application, you'll want to deploy it so that other people can use it. In this chapter, I'll introduce you to the options available for deploying your application to a production server, and walk you through the steps involved in taking your application to the world.

The Book's Web Site

Head over to http://www.sitepoint.com/books/rails2/ for easy access to various resources supporting this book.

The Code Archive

The code archive for this book, which can be downloaded from http://www.sitepoint.com/books/rails2/archive/, contains each and every line of example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the files.

Updates and Errata

While everyone involved in producing a technical book like this goes to enormous effort to ensure the accuracy of its content, books tend to have errors. Fortunately, the Corrections and Typos page located at http://www.sitepoint.com/books/rails2/errata.php is the most current, comprehensive reference for spelling and code-related errors that observant readers have reported to us.

The SitePoint Forums

If you have a problem understanding any of the discussion or examples in this book, try asking your question in the SitePoint Forums, at http://www.sitepoint.com/forums/. There, the enthusiastic and friendly community will be able to help you with all things Rails.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including *The SitePoint Tribune* and *The SitePoint Tech Times*. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development.

You can count on gaining some useful Rails articles and tips from these resources, but if you're interested in learning other technologies, or aspects of web development and business, you'll find them especially valuable. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you can't find your answer through the forums, or if you wish to contact us for any other reason, write to `books@sitepoint.com`. We have a well-staffed email support system set up to track your inquiries, and if our support staff members are unable to answer your question, they'll send it straight to me. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A perfect summer's day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code may be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me …

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always …

… pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Acknowledgments

Thanks to the great people at SitePoint for giving me the chance to write this book. In particular, thanks to Technical Editors Matthew Magain and Andrew Tetlaw for their crisp, sharp commentary, and Managing Editor Simon Mackie for applying an appropriate measure of brute force to me and my drafts—dedication that ensured that this book is in the best shape possible. I am truly grateful for this opportunity.

To the people in the Rails Core team, for making my developer life enjoyable again by putting together this amazing framework in almost no time, bringing outstanding improvements to an already great foundation, and laying the base on which this book could be written, thank you.

Special thanks to the makers of the Red Bull energy drink, without which the countless nights that went into both the first and second edition of this book wouldn't have been as productive as they were. In a related fashion, my sincere thanks to the rock bands 30 Seconds to Mars, Linkin Park, Fall Out Boy, Placebo, Billy Talent, and Good Charlotte, among others, for orchestrating the late-night writing sessions.

Finally, thanks must go to my family, especially Alice and Gwen, for giving me so much strength, motivation, and confidence in what I'm doing. Thank you for bearing the fact that I was rarely seen away from a computer for way too long. Many thanks to my dad, for the foundation of my professional life and for teaching me so many things that could fill an entire book on their own. Thank you!

Chapter 1

Introducing Ruby on Rails

Since Ruby on Rails was first released, it has become a household name (well, in developers' households, anyway). Hundreds of thousands of developers the world over have adopted—and adored—this new framework. I hope that, through the course of this book, you'll come to understand the reasons why. Before we jump into writing any code, let's take a stroll down memory lane, as we meet Ruby on Rails and explore a little of its history.

First, what exactly *is* Ruby on Rails?

The short—and fairly technical—answer is that **Ruby on Rails** (often abbreviated to "Rails") is a full-stack web application framework, written in Ruby. However, depending on your previous programming experience (and your mastery of techno-jargon), that answer might not make a whole lot of sense to you. Besides, the Ruby on Rails movement—the development principles it represents—really needs to be viewed in the context of web development in general if it is to be fully appreciated.

So, let's define a few of the terms mentioned in the definition above, and take in a brief history lesson along the way. Then we'll tackle the question of why learning Rails is one of the smartest moves you can make for your career as a web developer.

- A **web application** is a software application that's accessed using a web browser over a network. In most cases, that network is the Internet, but it could also be a corporate intranet. The number of web applications being created has increased exponentially since Rails was created, due mostly to the increased availability of broadband Internet access and the proliferation of faster desktop machines in people's homes. It can only be assumed that you're interested in writing such a web application, given that you've bought this book!

- A **framework** can be viewed as the foundation of a web application. It takes care of many of the low-level details that can become repetitive and boring to code, allowing the developer to focus on building the application's functionality.

  A framework gives the developer classes that implement common functions used in *every* web application, including:

  - database abstraction (ensuring that queries work regardless of whether the database is MySQL, Oracle, DB2, SQLite, or [insert your favorite database here])

  - templating (reusing presentational code throughout the application)

  - management of user sessions

  - generation of clean, search engine-friendly URLs

  A framework also defines the architecture of an application; this facility can be useful for those of us who fret over which file is best stored in which folder.

  In a sense, a framework is an application that has been started for you, and a well-designed application at that. The structure, plus the code that takes care of the boring stuff, has already been written, and it's up to you to finish it off!

- **Full-stack** refers to the extent of the functionality that the Rails framework provides. You see, there are frameworks and then there are frameworks. Some provide great functionality on the server, but leave you high and dry on the client side; others are terrific at enhancing the user experience on the client machine, but don't extend to the business logic and database interactions on the server.

  If you've ever used a framework before, chances are that you're familiar with the model-view-controller (MVC) architecture (if you're not, don't worry—we'll discuss it in Chapter 5). Rails covers *everything* in the MVC paradigm, from database abstraction to template rendering, and everything in between.

- **Ruby** is an open source, object oriented scripting language invented by Yukihiro Matsumoto in the early 1990s. We'll be learning both Ruby *and* Rails as we progress through the book (remember, Rails is written in Ruby).

  Ruby makes programming flexible and intuitive, and with it, we can write code that's readable by both humans and machines. Matsumoto clearly envisioned Ruby as a programming language that would entail very little mental overhead for humans, which is why Ruby programmers tend to be happy programmers.

What Does Ruby Look Like?

If you're experienced in programming with other languages, such as PHP or Java, you can probably make some sense of the following Ruby code, although some parts of it may look new:

01-ruby-sample.rb *(excerpt)*

```
>> "What does Ruby syntax look like?".reverse
=> "?ekil kool xatnys ybuR seod tahW"
>> 8 * 5
=> 40
>> 3.times { puts "cheer!" }
cheer!
cheer!
cheer!
>> %w(one two three).each { |word| puts word.upcase }
ONE
TWO
THREE
```

Don't worry too much about the details of programming in Ruby for now—we'll cover all of the Ruby basics in Chapter 3.

History

Ruby on Rails originated as an application named Basecamp,[1] a hosted project-management solution created by Danish web developer David Heinemeier Hansson for former design shop 37signals.[2] Due largely to Basecamp's success, 37signals has since moved into application development and production, and Heinemeier Hansson has become a partner in the company.

When I say "originated," I mean that Rails wasn't initially created as a stand-alone framework. It was extracted from a real application that was already in use, so that it could be used to build other applications that 37signals had in mind.[3] Heinemeier Hansson saw the potential to make his job (and life) easier by extracting common functionality such as database abstraction and templating into what later became the first public release of Ruby on Rails.

He decided to release Rails as open source software to "fundamentally remake the way web sites are built."[4] The first beta version of Rails was initially released in July 2004, with the 1.0 and 2.0 releases following on December 13, 2005 and December 07, 2007 respectively. Several hundreds of thousands of copies of Rails have been downloaded over time, and that number is climbing.

The fact that the Rails framework was extracted from Basecamp is considered by the lively Rails community to represent one of the framework's inherent strengths: Rails was already solving *real* problems when it was released. Rails wasn't built in isolation, so its success wasn't a result of developers taking the framework, building applications with it, and then finding—and resolving—its shortcomings. Rails had already proven itself to be a useful, coherent, and comprehensive framework.

While Heinemeier Hansson pioneered Rails and still leads the Rails-related programming efforts, the framework has benefited greatly from being released as open source software. Over time, developers working with Rails have submitted thousands of

[1] http://www.basecamphq.com/
[2] http://www.37signals.com/
[3] Highrise [http://www.highrisehq.com/], Backpack [http://www.backpackit.com/], Ta-da List [http://www.tadalist.com/], Campfire [http://www.campfirenow.com/], and Writeboard [http://www.writeboard.com/] are other hosted applications written in Rails by 37signals.
[4] http://www.wired.com/wired/archive/14.04/start.html?pg=3

extensions and bug fixes to the Rails development repository.[5] The repository is closely guarded by the Rails core team, which consists of about six highly skilled professional developers chosen from the crowd of contributors, led by Heinemeier Hansson.

So, now you know what Rails is, and how it came about. But why would you invest your precious time in learning how to use it?

I'm glad you asked.

Development Principles

Rails supports several software principles that make it stand out from other web development frameworks. Those principles are:

- convention over configuration
- don't repeat yourself
- agile development

Because of these principles, Ruby on Rails is a framework that really does save developers time and effort. Let's look at each of those principles in turn to understand how.

Convention Over Configuration

The concept of **convention over configuration** refers to the fact that Rails assumes a number of defaults for the way one should build a typical web application.

Many other frameworks (such as the Java-based Struts or the Python-based Zope) require you to step through a lengthy configuration process before you can make a start with even the simplest of applications. The configuration information is usually stored in a handful of XML files, and these files can become quite large and cumbersome to maintain. In many cases, you're forced to repeat the entire configuration process whenever you start a new project.

While Rails was originally extracted from an existing application, extensive architectural work went into the framework later on. Heinemeier Hansson purposely

[5] The Rails repository, located at http://dev.rubyonrails.org/, is used to track bugs and enhancement requests.

created Rails in such a way that it doesn't need excessive configuration, as long as some standard conventions are followed. The result is that no lengthy configuration files are required. In fact, if you have no need to change these defaults, Rails really only needs a single (and short) configuration file in order to run your application. The file is used to establish a database connection: it supplies Rails with the necessary database server type, server name, user name, and password for each environment, and that's it. Here is an example of a configuration file (we'll talk more about the contents of this configuration file in Chapter 4):

02-database.yml

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  timeout: 5000
test:
  adapter: sqlite3
  database: db/test.sqlite3
  timeout: 5000
production:
  adapter: sqlite3
  database: db/production.sqlite3
  timeout: 5000
```

Other conventions that are prescribed by Rails include the naming of database-related items, and the process by which **controllers** find their corresponding **models** and **views**.

Controllers? Models? Views? *Huh?*

Model-view-controller (MVC) is a software architecture (also referred to as a design pattern) that separates an application's **data model** (model), **user interface** (view), and **control logic** (controller) into three distinct components.

Here's an example: when your browser requests a web page from an MVC-architected application, it's talking exclusively to the controller. The controller gathers the required data from one or more models and renders the response to your request through a view. This separation of components means that any change that's made to one component has a minimal effect on the other two.

We'll talk at length about the MVC architecture and the benefits it yields to Rails applications in Chapter 5.

Rails is also considered to be **opinionated software**, a term that has been coined to refer to software that isn't everything to everyone. Heinemeier Hansson and his core team ruthlessly reject contributions to the framework that don't comply with their vision of where Rails is headed, or aren't sufficiently applicable to be useful for the majority of Rails developers. This is a good way to fight a phenomenon known among software developers as **bloat**: the tendency for a software package to implement extraneous features just for the sake of including them.

Don't Repeat Yourself

Rails supports the principles of **DRY** (Don't Repeat Yourself) programming. When you decide to change the behavior of an application that's based on the DRY principle, you shouldn't need to modify application code in more than one authoritative location.

While this might sound complicated, it's actually quite simple. For example, instead of copying and pasting code with a similar or even identical functionality, you develop your application in such a way that this functionality is stored once, in a central location, and is referenced from each portion of the application that needs to use it. This way, if the original behavior needs to change, you need only make modifications in one location, rather than in various places throughout your application—some of which you could all too easily overlook.

One example of how Rails supports the DRY principle is that, unlike Java, it doesn't force you to repeat your **database schema definition** within your application. A database schema definition describes how the storage of an application's data is structured. Think of it as a number of spreadsheets, each of which contains rows and columns that define the various pieces of data, and identify where each data item is stored. Rails considers your database to be the authoritative source of information about data storage, and is clever enough to ask the database for any information it may need to ensure that it treats your data correctly.

Rails also adheres to the DRY principle when it comes to implementing cutting-edge techniques such as **Ajax** (Asynchronous JavaScript and XML). Ajax is an approach that allows your web application to replace content in the user's browser dynamically, or to exchange form data with the server without reloading the page. Developers often find themselves duplicating code while creating Ajax applications: after all, the web site should function in browsers that *don't* support Ajax, as well as those that do, and the code required to display the results to both types of browser is, for the most part, identical. Rails makes it easy to treat each browser generation appropriately without duplicating any code.

Agile Development

More traditional approaches to software development (such as iterative development and the waterfall model) usually attempt to sketch out a long-running and rather static plan for an application's goals and needs using predictive methods. These development models usually approach applications from the bottom up—that is, by working on the data first.

In contrast, **Agile** development methods use an **adaptive** approach. Small teams, typically consisting of fewer than ten developers, iteratively complete small units of the project. Before starting an iteration, the team reevaluates the priorities for the application that's being built; these priorities may have shifted during the previous iteration, so they may need adjustment. Agile developers also architect their applications from the top down, starting with the design, which may be as simple as a sketch of the interface on a sheet of paper.

When an application is built using Agile methods, it's less likely to veer out of control during the development cycle, thanks to the ongoing efforts of the team to adjust priorities. By spending less time on the creation of functional specifications and long-running schedules, developers using Agile methodologies can really jump-start an application's development.

Here are a few examples that illustrate how Rails lends itself to Agile development practices:

- You can start to work on the layout of your Rails application before making any decisions about data storage (even though these decisions might change at a later stage). You don't have to repeat this layout work when you start adding functionality to your screen designs—everything evolves dynamically with your requirements.

- Unlike code written in C or Java, Rails applications don't need to go through a compilation step in order to be executable. Ruby code is interpreted on the fly, so it doesn't need any form of binary compilation to make it executable. Changing code during development provides developers with immediate feedback, which can significantly boost the speed of application development.

- Rails provides a comprehensive framework for the automated testing of application code. Developers who make use of this testing framework can be confident that they're not causing functionality to break when they change existing code, even if they weren't the ones who originally developed it.

- Refactoring (rewriting code with an emphasis on optimization) existing Rails application code to better cope with changed priorities, or to implement new features for a development project, can be done much more easily when developers adhere to the DRY principles we discussed above. This is because far fewer changes are required when a certain functionality is implemented just once, and is then reused elsewhere as required.

If your head is spinning from trying to digest these principles, don't worry—they'll be reinforced continually throughout this book, as we step through building our very own web application in Ruby on Rails!

Building the Example Web Application

As you read on, I expect you'll be itching to put the techniques we discuss into practice. For this reason, I've planned a fully functional web application that we'll build together through the ensuing chapters. The key concepts, approaches, and methodologies we'll discuss will have a role to play in the sample application, and we'll implement them progressively as your skills improve over the course of this book.

The application we'll build will be a functional clone of the popular story-sharing web site, Digg.[6] I've included all necessary files for this application in this book's code archive.

[6] http://www.digg.com/

What Is Digg?

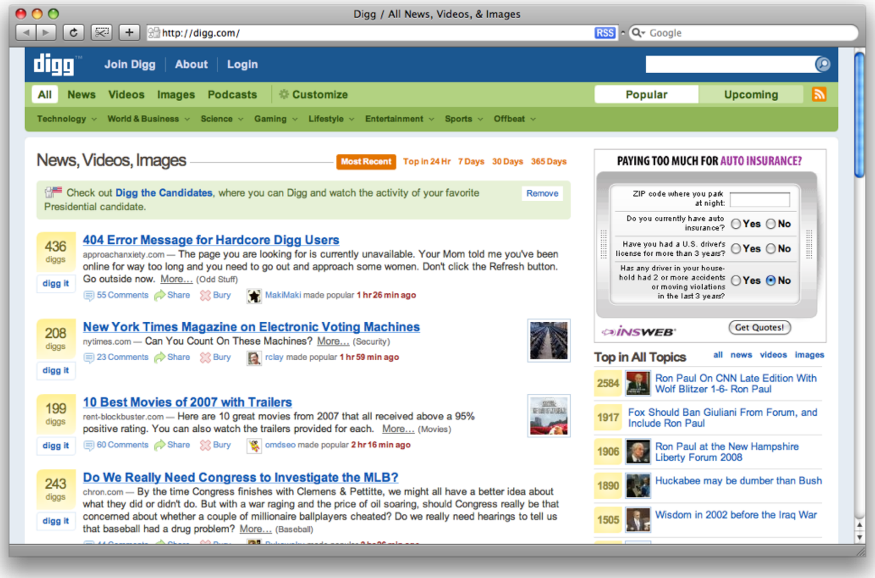
Digg describes itself as follows:[7]

> Digg is a place for people to discover and share content from anywhere on the web. From the biggest online destinations to the most obscure blog, Digg surfaces the best stuff as voted on by our users. You won't find editors at Digg—we're here to provide a place where people can collectively determine the value of content and we're changing the way people consume information online.
>
> How do we do this? Everything on Digg—from news to videos to images to podcasts—is submitted by our community (that would be you). Once something is submitted, other people see it and Digg what they like best. If your submission rocks and receives enough Diggs, it is promoted to the front page for the millions of our visitors to see.

Basically, if you want to tell the world about that interesting article you found on the Internet—be it a weblog post that's right up your street, or a news story from a major publication—you can submit its URL to Digg, along with a short summary of the item. Your story sits in a queue, waiting for other users to **digg** it (give your item a positive vote). As well as voting for a story, users can comment on the story to create often lively discussions within Digg.

As soon as the number of diggs for a story crosses a certain threshold, it's automatically promoted to the Digg homepage, where it attracts a far greater number of readers than the story-queuing area receives. Figure 1.1 shows a snapshot of the Digg homepage.

[7] http://digg.com/about/



News, Videos, Images

Check out Digg the Candidates, where you can Digg and watch the activity of your favorite Presidential candidate.

436 diggs — 404 Error Message for Hardcore Digg Users
approachanxiety.com — The page you are looking for is currently unavailable. Your Mom told me you've been online for way too long and you need to go out and approach some women. Don't click the Refresh button. Go outside now. More... (Odd Stuff)
55 Comments · MakiMaki made popular 1 hr 26 min ago

208 diggs — New York Times Magazine on Electronic Voting Machines
nytimes.com — Can You Count On These Machines? More... (Security)
23 Comments · rclay made popular 1 hr 59 min ago

199 diggs — 10 Best Movies of 2007 with Trailers
rent-blockbuster.com — Here are 10 great movies from 2007 that all received above a 95% positive rating. You can also watch the trailers provided for each. More... (Movies)
60 Comments · omdseo made popular 2 hr 16 min ago

243 diggs — Do We Really Need Congress to Investigate the MLB?
chron.com — By the time Congress finishes with Clemens & Pettitte, we might all have a better idea about what they did or didn't do. But with a war raging and the price of oil soaring, should Congress really be that concerned about whether a couple of millionaire ballplayers cheated? Do we really need hearings to tell us that baseball had a drug problem? More... (Baseball)

PAYING TOO MUCH FOR AUTO INSURANCE?

ZIP code where you park at night:
Do you currently have auto insurance? Yes No
Have you had a U.S. driver's license for more than 3 years? Yes No
Has any driver in your household had 2 or more accidents or moving violations in the last 3 years? Yes [✓ No]

Top in All Topics

2584	Ron Paul On CNN Late Edition With Wolf Blitzer 1-6- Ron Paul
1917	Fox Should Ban Giuliani From Forum, and Include Ron Paul
1906	Ron Paul at the New Hampshire Liberty Forum 2008
1890	Huckabee may be dumber than Bush
1505	Wisdom in 2002 before the Iraq War

Figure 1.1. The original digg.com

The Digg Effect

Due to the huge number of visitors that Digg receives, web sites that are listed on the front page may suffer from what is known as the **Digg effect**: the servers of many sites cannot cope with the sudden surge in traffic, and become inaccessible until the number of simultaneous visitors dies down or the hosting company boosts the site's capacity to deal with the increase in traffic.

Digg was launched in December 2004, and has since been listed in the Alexa traffic rankings as one of the Internet's top 200 web sites.[8]

I didn't decide to show you how to develop your own Digg clone just because the site is popular with Internet users, though; Digg's feature set is not particularly complicated, but it's sufficient to allow us to gain firsthand experience with the most important and useful facets of the Ruby on Rails framework.

And while your application might not be able to compete with the original site, re-using this sample project to share links within your family, company, or college

[8] http://www.alexa.com/data/details/traffic_details/digg.com

class is perfectly conceivable. Also, with any luck you'll learn enough along the way to branch out and build other types of applications as well.

Features of the Example Application

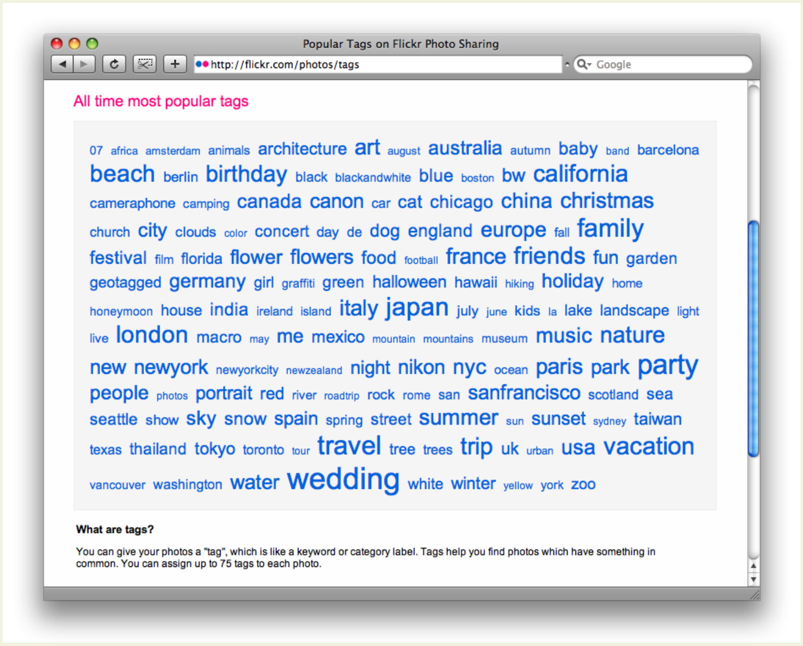
As I mentioned, we want our application to accept user-submitted links to stories on the Web. We also want to allow other users to vote on the submitted items. In order to meet these objectives, we'll implement the following features as we work through this book:

- We'll build a database back end that permanently stores every story, user, vote, and so on. This way, nothing is lost when you close your browser and shut the application down.
- We'll build a story submission interface, which is a form that's available only to users who have registered and logged in.
- We'll develop a simplistic layout, as is typical for Web 2.0 applications. We'll style it with Cascading Style Sheets (CSS) and enhance it with visual effects.
- We'll create clean URLs for all the pages on our site. **Clean URLs** (also known as search engine-friendly URLs) are usually brief and easily read when they appear in the browser status bar. An example of a clean URL is http://del.icio.us/popular/software, which I'm sure you'll agree is a lot nicer than http://www.amazon.com/gp/homepage.html/103-0615814-1415024/.
- We'll create a user registration system that allows users to log in with their usernames and passwords.
- We'll create two different views for stories: the homepage of our application, and the story queue containing stories that haven't yet received enough votes to appear on the homepage.
- We'll give users the ability to check voting history on per-user and per-story bases.
- We'll facilitate the tagging of stories, and give users the ability to view only those stories that relate to **`programming`** or **`food`**, for example. For a definition of tagging, see the note below.

It’s quite a list, and the result will be one slick web application! Some of the features rely upon others being in place, and we’ll implement each feature as a practical example when we look at successive aspects of Rails.

What is Tagging?

Tagging can be thought of as a free-form categorization method. Instead of the site’s owners creating a fixed content categorization scheme (often represented as a tree), users are allowed to enter one or more keywords to describe a content item. Resources that share one or more identical tags can be linked together easily—the more overlap between tags, the more characteristics the resources are likely to have in common.


Popular Tags on Flickr Photo Sharing
http://flickr.com/photos/tags
Google

All time most popular tags

07 africa amsterdam animals architecture art august australia autumn baby band barcelona beach berlin birthday black blackandwhite blue boston bw california cameraphone camping canada canon car cat chicago china christmas church city clouds color concert day de dog england europe fall family festival film florida flower flowers food football france friends fun garden geotagged germany girl graffiti green halloween hawaii hiking holiday home honeymoon house india ireland island italy japan july june kids la lake landscape light live london macro may me mexico mountain mountains museum music nature new newyork newyorkcity newzealand night nikon nyc ocean paris park party people photos portrait red river roadtrip rock rome san sanfrancisco scotland sea seattle show sky snow spain spring street summer sun sunset sydney taiwan texas thailand tokyo toronto tour travel tree trees trip uk urban usa vacation vancouver washington water wedding white winter yellow york zoo

What are tags?

You can give your photos a "tag", which is like a keyword or category label. Tags help you find photos which have something in common. You can assign up to 75 tags to each photo.

Figure 1.2. Tags on flickr.com

Instead of displaying a hierarchical category tree, the tags used in an application are commonly displayed as a **tag cloud** in which each of the tags is represented in a font size that corresponds to how often that tag has been applied to content items within the system.

Tags are used extensively on sites such as the Flickr photo-sharing web site[9] shown in Figure 1.2, the del.icio.us bookmark-sharing site,[10] and the Technorati weblog search engine.[11]

Summary

We've explored a bit of history in this chapter. Along the way, we learned where both the Ruby language and the Rails framework have come from, and looked in some detail at the niche that followers of the Ruby on Rails philosophy have carved out for themselves in the web development world. I also explained the philosophy behind the Ruby programming language and showed you a snippet of Ruby code. We'll cover much more of Ruby's inner workings in Chapter 3.

We also talked briefly about some of the basic principles that drive Rails development, and saw how Rails supports Agile development methods. Now that you're aware of the possibilities, perhaps some of these ideas and principles will influence your own work with Rails.

Finally, we created a brief specification for the web application we're going to build throughout this book. We described what our application will do, and identified the list of features that we're going to implement. We'll develop a clone of the story-sharing web site Digg iteratively, taking advantage of some of the Agile development practices that Rails supports.

In the next chapter, we'll install Ruby, Rails, and the SQLite database server software in order to set up a development environment for the upcoming development tasks.

Are you ready to join in the fun? If so, turn the page …

[9] http://flickr.com/
[10] http://del.icio.us/
[11] http://www.technorati.com/

Chapter 2

Getting Started

To get started with Ruby on Rails, we first need to install some development software on our systems. The packages we'll be installing are:

the Ruby language interpreter
: The Ruby interpreter translates our Ruby code (or any Ruby code, for that matter, including Rails itself) into a form the computer can understand and execute. At the time of writing, Ruby 1.8.6 is recommended for use with Rails, so that's what I've used here.

the Ruby on Rails framework
: Once we've downloaded Ruby, we can install the Rails framework itself. As I mentioned in Chapter 1, Rails is written in Ruby. At the time of writing, version 2.0.2 was the most recent stable version of the framework.

the SQLite database engine
: The SQLite database engine is a self-contained software library which provides an SQL database without actually running a separate server process. While Rails supports plenty of other database servers (MySQL, PostgreSQL, Microsoft SQL Server, and Oracle, to name a few), SQLite is easy to install and does not require

any configuration, and is the default database for which a new Rails application is configured straight out of the box. Oh, and it's free!

At the time of writing, the most recent stable release of the SQLite database was version 3.5.4.

Instructions for installing Rails differ ever so slightly between operating systems. You may also need to install some additional tools as part of the process, depending on the platform you use. Here, I'll provide installation instructions for Windows, Mac OS X, and Linux.

Watch Your Version Numbers!

It's possible that by the time you read this, a more recent version of Ruby, SQLite, or one of the other packages mentioned here will have been released. Beware! Don't just assume that because a package is newer, it can reliably be used for Rails development. While, in theory, every version should be compatible and these instructions should still apply, sometimes the latest is *not* the greatest.

In fact, the Rails framework itself also has a reputation for experiencing large changes between releases, such as specific methods or attributes being deprecated. While every effort has been made to ensure the code in this book is future-proof, there's no guarantee that changes included in future major releases of Rails won't require this code to be modified in some way for it to work. Such is the fast-paced world of web development!

Feel free to skip the sections relevant to operating systems other than yours, and to focus on those that address your specific needs.

What Does All This Cost?

Everything we need is available for download from the Web, and is licensed under free software licenses. This basically means that everything you'll be installing is free for you to use in both personal and commercial applications. If you're curious about the differences between each license, you can check out each package's individual license file, which is included in its download.

Installing on Windows

For some reason, Windows has the easiest install procedure. A very helpful programmer by the name of Curt Hibbs sat down and packaged everything required to develop Rails applications on a Windows machine. He constructed the package as an easy-to-install, easy-to-run, single file download called Instant Rails and released version 1.0 in early 2006. When Rails 2.0 was released in late 2007, maintenance of Instant Rails was taken over by Rob Bazinet, and nowadays consists of the following components:

- the Ruby interpreter
- the SQLite and MySQL database engines
- the Apache web server (although we won't be using it in this book)
- Ruby on Rails

That's everything we need in one handy package. How convenient!

To install Instant Rails, download the latest Instant Rails zip archive from the Instant Rails project file list[1] on RubyForge and extract its contents to a folder of your choice.

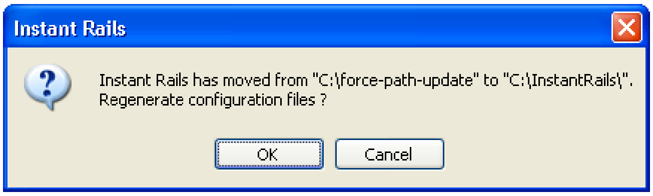
Time Flies

The version of Instant Rails used to test the code in this book was 2.0. As discussed earlier in the chapter, due to the fast-changing nature of the framework, I can't guarantee that later versions will work.

Be careful, though; Instant Rails doesn't support folders with names that contain spaces; unfortunately, this means that the obvious choice of **C:\Program Files** is not a good one. I recommend choosing **C:\InstantRails\** instead.

After you've extracted the **.zip** file (it has approximately 18,000 items in packaged documentation, so if you're using the Windows built-in file compression tool, it could take quite some time to unzip them all), navigate to the **InstantRails** folder and double-click the **InstantRails.exe** file. You'll be prompted with a dialog like the one shown in Figure 2.1; click **OK** to continue.

[1] http://rubyforge.org/frs/?group_id=904

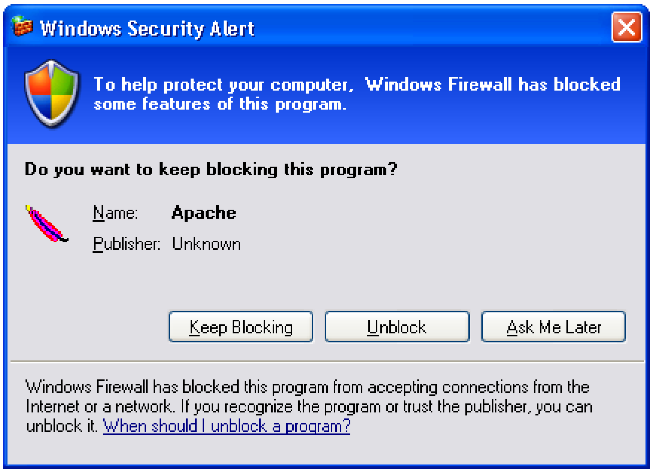


Instant Rails

Instant Rails has moved from "C:\force-path-update" to "C:\InstantRails\".
Regenerate configuration files ?

Figure 2.1. Configuring Instant Rails doesn't get much easier …

If you're on Windows XP Service Pack 2 or later, you'll also be greeted with the alert message in Figure 2.2 from the Windows internal firewall (or any additional personal firewall software that you might have installed). Of course, the Apache web server isn't trying to do anything malicious—Instant Rails just fires it up as part of its initialization process. Go ahead and click **Unblock** to allow it to start.



Windows Security Alert

To help protect your computer, Windows Firewall has blocked some features of this program.

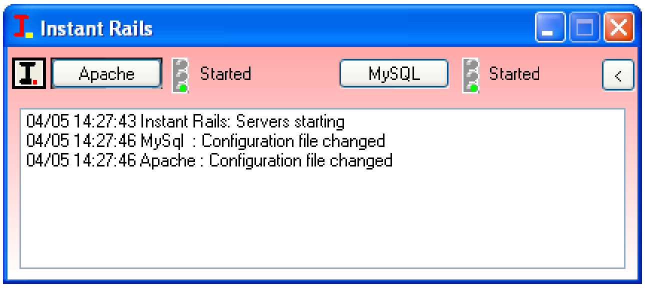
Do you want to keep blocking this program?

Name: Apache
Publisher: Unknown

Windows Firewall has blocked this program from accepting connections from the Internet or a network. If you recognize the program or trust the publisher, you can unblock it. When should I unblock a program?

Figure 2.2. Allowing Apache to run

You should now see the Instant Rails control panel, which, as Figure 2.3 illustrates, should report that everything has started up successfully.



```
04/05 14:27:43 Instant Rails: Servers starting
04/05 14:27:46 MySql  : Configuration file changed
04/05 14:27:46 Apache : Configuration file changed
```

Figure 2.3. The Instant Rails control panel

Next, you'll need to update the version of RubyGems that comes with Instant Rails. Click on the `I` button in the top-left of the control panel. From the menu that appears, select **Rails Applications** > **Open Ruby Console Window**. Once the console opens, enter the following command:

```
C:\InstantRails\rails_apps> gem update --system
```

The final step is to update Rails. The following command, entered in the Ruby Console you opened previously, should do the trick:

```
C:\InstantRail\rails_apps> gem update
```

That's it! Everything you need is installed and configured. Feel free to skip the instructions for Mac and Linux, and start building your application!

Installing on Mac OS X

Okay, so the Windows guys had it easy. Unfortunately, life isn't quite so simple for the rest of us, at least as far as Rails installation is concerned. While Mac OS X isn't usually a platform that makes things tricky, installing Rails is just a tad harder than installing a regular Mac application.

Lose the Locomotive

There was an all-in-one installer available for Mac OS X 10.4 and earlier, called Locomotive (http://locomotive.raaum.org/). Unfortunately, it's no longer maintained by its creator, Ryan Raaum. For this reason, I don't recommend that you use Locomotive to work through this book.

Mac OS X 10.5 (Leopard)

If your Mac is a relatively recent purchase, you may be running OS X version 10.5 (Leopard) or later. If this is the case, you've got much less to do, because your machine comes preinstalled with both Ruby *and* Rails—congratulations! All you'll need to do is update your Rails installation, but we'll worry about that when we reach that step in the Rails installation instructions for Mac OS X 10.4—you'll see that it's very easy.

Of course, it wouldn't do you any harm to read through all of the steps below anyway, just to make sure that you're familiar with the software components and concepts that are introduced; for example, the Mac OS X Terminal, the command line interface, and RubyGems.

Mac OS X 10.4 (Tiger) and Earlier

"But wait!" I hear you cry. "My slightly older Mac comes with Ruby preinstalled!" Yes, that may indeed be true. However, the version of Ruby that shipped with OS X prior to version 10.5 is a slimmed-down version that's incompatible with Rails, and is therefore unsuited to our needs. While there are packages out there that make the installation of Ruby easier, such as MacPorts,[2] for the sake of completeness, I'll show you how to build Ruby on your machine from scratch.[3] Don't worry. It may sound intimidating, but it's actually relatively painless—and you'll only need to do it once!

Let's start installing then, shall we?

Installing Xcode

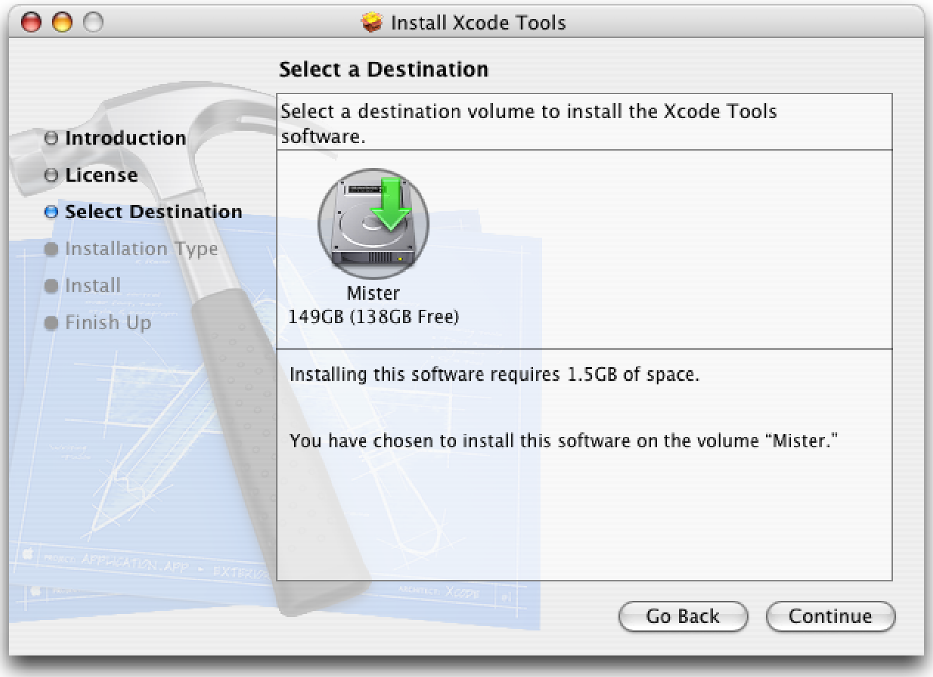
The first step in the process is to make sure we have everything we need for the installation to go smoothly. The only prerequisite for this process is Xcode, the Apple Developer Tools that come on a separate CD with Mac OS X. If you haven't

[2] http://www.macports.org/

[3] A tip of the hat is in order for Dan Benjamin, who did a lot of the heavy lifting in the early days of documenting the installation of Rails on OS X. Parts of these installation instructions are heavily influenced by his article "Building Ruby, Rails, Subversion, Mongrel, and MySQL on Mac OS X". [http://hivelogic.com/articles/ruby-rails-mongrel-mysql-osx/]

installed the tools yet, and don't have your installation CD handy, you can download the Xcode package for free[4] (although at more than 900MB, it's a hefty download!).

To install Xcode, run the packaged installer by clicking on the **XcodeTools.mpkg** icon and following the on-screen instructions illustrated in Figure 2.4. The installation tool is a simple wizard that will require you to click **Continue** a few times, agree to some fairly standard terms and conditions, and hit the **Install** button.



Select a destination volume to install the Xcode Tools software.

Installing this software requires 1.5GB of space.

You have chosen to install this software on the volume "Mister."

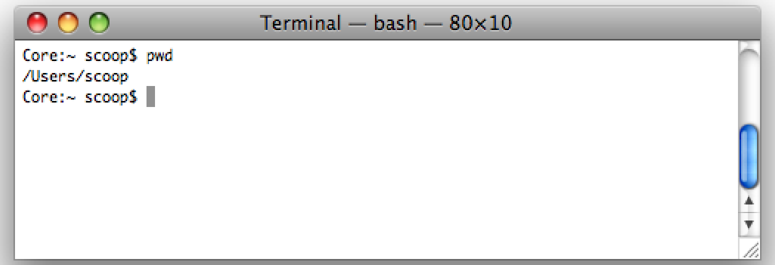
Figure 2.4. Installing the Xcode developer tools

Introducing the Command Line

For the next few steps, we're going to leave the comfort and security of our pretty graphical user interface and tackle the much geekier UNIX command line. If this is the first time you've used the command line on your Mac, don't worry—we'll be using it often as we work through this book, so you'll have plenty of practice! Let's dive in.

[4] http://developer.apple.com/

First, open up a UNIX session in OS X using the Terminal utility. Launch Terminal by selecting **Go** > **Utilities** from the Finder menu bar, and double-clicking the **Terminal** icon. Your Terminal window should look a lot like Figure 2.5.



```
Core:~ scoop$ pwd
/Users/scoop
Core:~ scoop$
```

Figure 2.5. A Terminal window on Mac OS X

Let's dissect these crazy command line shenanigans. The collection of characters to the left of the cursor is called the **prompt**. By default, it displays:

- the name of your Mac
- the current directory
- the name of the user who's currently logged in

In my case, this is:

```
Core:~ scoop$
```

So, what's what here?

- `Core` is the name of my Mac.
- `scoop` is the name of the user who's currently logged in.

But what on earth is *this* character: ~? It's called a **tilde**, and it's shorthand notation for the path to the current user's home directory. Take another look at Figure 2.5, and you'll see I've used the `pwd` command to *p*rint the *w*orking *d*irectory. The result is **/Users/scoop**, which just happens to be my home directory. For future command line instructions, though, I'll simply display the prompt as: `$`, to avoid occupying valuable real estate on the page.

Setting the Path

Next, we need to make sure that Mac OS X can locate all the command line tools that we'll be using during this installation. The `PATH` environment variable stores the list of folders to which OS X has access; we'll store the changes we make to this variable in a file in our home directory.

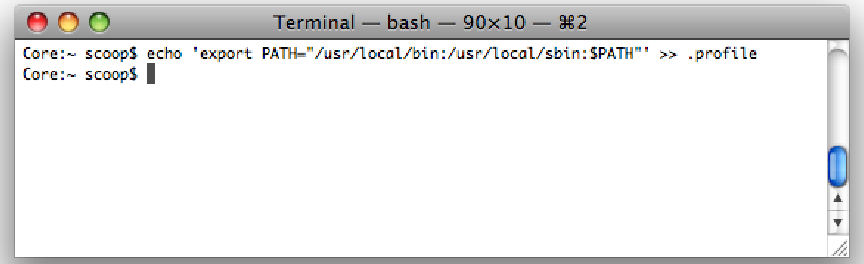
The name of the file we'll use is **.profile**. (On UNIX-based systems such as Mac OS X, files that start with a period are usually hidden files.) If you type out the following command exactly, your PATH environment variable will be set correctly every time you open a new Terminal window:

```
$ echo 'export PATH="/usr/local/bin:/usr/local/sbin:$PATH"' >>
➥    ~/.profile
```

To activate this change (without having to open and close the Terminal window), type the following command:

```
$ . ~/.profile
```

Yes, that's a single period at the beginning of the line. Note that these commands don't produce any feedback, as Figure 2.6 shows, but they're still taking effect.



```
Core:~ scoop$ echo 'export PATH="/usr/local/bin:/usr/local/sbin:$PATH"' >> .profile
Core:~ scoop$ 
```

Figure 2.6. Setting the correct path

It would be a shame to clutter up this home directory—or our desktop—with a huge number of files, so let's go about this installation business in an organized fashion.

Staying Organized

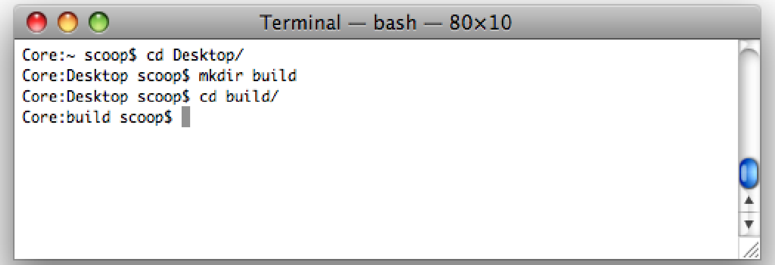
The process of extracting, configuring, and compiling the source code for all the packages that we'll be downloading will take up a reasonable amount of space on your hard drive. Let's keep things organized and operate within a single folder rather than making a mess of the desktop.

The desktop on your Mac is actually a subfolder of your home directory. Change to the **Desktop** folder using the `cd` command (short for *c*hange *d*irectory). Once you're there, you can use `mkdir` to *make* a *dir*ectory in which to store our downloads and other assorted files. Let's call this directory **build**:

```
$ cd Desktop
$ mkdir build
$ cd build
```

The result is that we have on the desktop a new directory, which is now our current working directory. This is also reflected by our prompt. As Figure 2.7 shows, mine now reads:

```
Core:build scoop$
```



```
Core:~ scoop$ cd Desktop/
Core:Desktop scoop$ mkdir build
Core:Desktop scoop$ cd build/
Core:build scoop$
```

Figure 2.7. Creating a temporary working directory

Now the fun begins!

Installing Ruby on a Mac

Before installing Ruby itself, we need to install another library on which Ruby depends: Readline.

Here's the sequence of slightly convoluted commands for installing the Readline library. It's a fairly small library, so the installation shouldn't take long:

```
$ curl ftp://ftp.gnu.org/gnu/readline/readline-5.2.tar.gz
➥ | tar xz
$ cd readline-5.2
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
$ cd ..
```

With Readline in place, we're now able to install Ruby itself. This step may test your patience a bit, as the configuration step could take half an hour or more to complete, depending on the speed of your system and your network connection. Type out the following series of commands, *exactly* as you see them here (it's not important that you understand every line, but it is important that you don't make any typos):

```
$ curl ftp://ftp.ruby-lang.org/pub/ruby/1.8/
➥ruby-1.8.6.tar.gz | tar xz
$ cd ruby-1.8.6
$ ./configure --prefix=/usr/local --enable-pthread
➥ --with-readline-dir=/usr/local
$ make
$ sudo make install
$ sudo make install-doc
$ cd ..
```

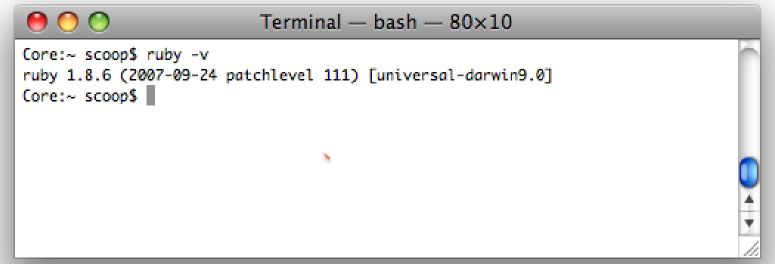
How did you do? It's wise to run some checks at this point, to determine whether our installation is on track so far. The simplest and safest way to ascertain whether our Ruby installation is working is to type the following command into the Terminal window:

```
$ ruby -v
```

The version displayed should match that which you downloaded—in my case, `ruby 1.8.6 (2007-09-24 patchlevel 111)`, as shown in Figure 2.8. If anything else is displayed here (such as `ruby 1.8.2 (2004-12-25)`), something's gone wrong. You should carefully repeat the instructions up to this point.

A Friend in Need's a Friend Indeed

Remember—if you get really stuck, you can always try asking for help on SitePoint's Ruby forum.[5]



```
Core:~ scoop$ ruby -v
ruby 1.8.6 (2007-09-24 patchlevel 111) [universal-darwin9.0]
Core:~ scoop$
```

Figure 2.8. Checking the Ruby version

Be a Super User for a Day

`sudo` is a way for "regular" computer users to perform system-wide installations that are normally reserved for system administrators. You'll need to enter your account password before you'll be allowed to execute this command. To use `sudo`, the user account must have the **Allow user to administer this computer** setting checked. This can be changed in the **Accounts** section of the Apple **System Preferences** window.

Next up, we have the installation of RubyGems.

Installing RubyGems on a Mac

"What is this RubyGems?" I hear you ask. RubyGems is a utility for managing the additions to the Ruby programming language that other people have developed and

[5] http://www.sitepoint.com/launch/rubyforum/

made available as free downloads. Think of it as prepackaged functionality that you can install on your machine so you don't have to reinvent the wheel over and over again while you're working on your own projects.[6] Rails is released and published through the RubyGems system.

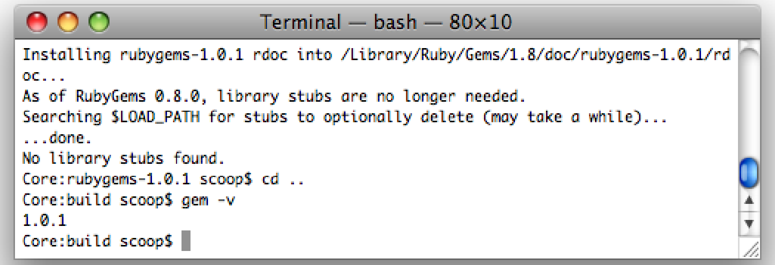
The following sequence of commands will download and install RubyGems on your Mac. It should be a relatively quick procedure:

```
$ curl -L http://rubyforge.org/frs/download.php/29548/
➥rubygems-1.0.1.tgz | tar xz
$ cd rubygems-1.0.1
$ sudo ruby setup.rb
$ cd ..
```

Are you getting the hang of this command line thing? Good! We now have *another* new command at our fingertips: `gem`. The `gem` command is used to install and manage Ruby packages on your machine—enter the following to check that Ruby-Gems is working properly:

```
$ gem -v
```

The output should identify the version of RubyGems that you installed, as Figure 2.9 shows. We'll use the `gem` command to install Rails.



```
Installing rubygems-1.0.1 rdoc into /Library/Ruby/Gems/1.8/doc/rubygems-1.0.1/rd
oc...
As of RubyGems 0.8.0, library stubs are no longer needed.
Searching $LOAD_PATH for stubs to optionally delete (may take a while)...
...done.
No library stubs found.
Core:rubygems-1.0.1 scoop$ cd ..
Core:build scoop$ gem -v
1.0.1
Core:build scoop$
```

Figure 2.9. Confirmation of a successful RubyGems installation

[6] The RubyGems web site [http://gems.rubyforge.org/] has additional documentation for the `gem` command that we'll use in this section.

Updating RubyGems

RubyGems is constantly being developed and improved, and new versions are released frequently. If you ever need to update RubyGems—for compatibility reasons, for example—simply enter the following command into a terminal window:

```
$ sudo gem update --system
```

Installing Rails on a Mac

Whew! After several pages of installation instructions, we're finally here: the installation of the Rails framework. Don't underestimate the importance of the work you've already done, though. Plus, you'll now have an easy upgrade path the next time a new version of Ruby or Rails is released (which I'll explain later).

Without further ado, enter this command to install Rails:

```
$ sudo gem install rails
```

This command prompts the RubyGems system to download Rails and the packages on which it depends, before installing the necessary files and documentation. This process may take ten minutes or more to complete, so it's a good time to grab yourself a coffee.

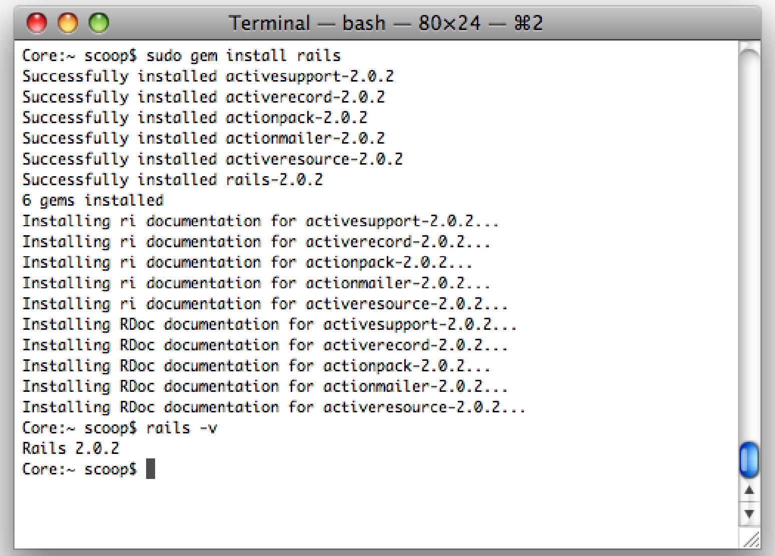
The Secret of Staying Up to Date with Rails

While you're waiting for Rails to install, I'll let you in on a little secret: the command we just entered is the very same one that you can use to stay up to date with future Rails releases. Whenever you want to upgrade Rails, just enter that command again, and your system will be updated. Very cool, as I'm sure you'll agree. In fact, if you're running the above command on Mac OS X 10.5, that's exactly what you'll be doing: upgrading your existing Rails installation.

Once the installation has finished, you can verify the version of Rails you just installed by running the following command in your Terminal window:

```
$ rails -v
```

Taking this small step should reward you with the version number of Rails, as illustrated in Figure 2.10.



```
Core:~ scoop$ sudo gem install rails
Successfully installed activesupport-2.0.2
Successfully installed activerecord-2.0.2
Successfully installed actionpack-2.0.2
Successfully installed actionmailer-2.0.2
Successfully installed activeresource-2.0.2
Successfully installed rails-2.0.2
6 gems installed
Installing ri documentation for activesupport-2.0.2...
Installing ri documentation for activerecord-2.0.2...
Installing ri documentation for actionpack-2.0.2...
Installing ri documentation for actionmailer-2.0.2...
Installing ri documentation for activeresource-2.0.2...
Installing RDoc documentation for activesupport-2.0.2...
Installing RDoc documentation for activerecord-2.0.2...
Installing RDoc documentation for actionpack-2.0.2...
Installing RDoc documentation for actionmailer-2.0.2...
Installing RDoc documentation for activeresource-2.0.2...
Core:~ scoop$ rails -v
Rails 2.0.2
Core:~ scoop$ 
```

Figure 2.10. Installing Rails on Mac OS X via RubyGems

I told you we'd get there in the end. Don't break out the champagne just yet, though—we still need a database!

Installing SQLite on a Mac

The last thing we need to do is install the storage container that's going to house the data we (or our users) enter through our application's web interface, the SQLite database engine.

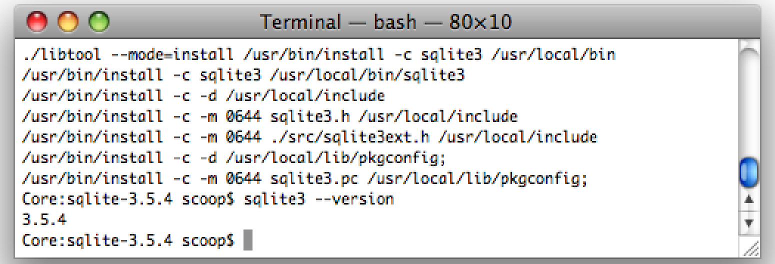
As mentioned in the introductory paragraph, SQLite is a self-contained, serverless database engine. As such, there's not much more to it than the actual download and compilation steps. You don't need to mess with startup scripts or configuration files. It all just works!

```
$ curl http://www.sqlite.org/sqlite-3.5.4.tar.gz | tar zx
$ cd sqlite-3.5.4
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

After the installation has been completed, enter the following command to make sure SQLite has been properly installed:

```
$ sqlite3 --version
```

This command should come back with the version of SQLite you installed—3.5.4 in my case—as shown in Figure 2.11.



```
./libtool --mode=install /usr/bin/install -c sqlite3 /usr/local/bin
/usr/bin/install -c sqlite3 /usr/local/bin/sqlite3
/usr/bin/install -c -d /usr/local/include
/usr/bin/install -c -m 0644 sqlite3.h /usr/local/include
/usr/bin/install -c -m 0644 ./src/sqlite3ext.h /usr/local/include
/usr/bin/install -c -d /usr/local/lib/pkgconfig;
/usr/bin/install -c -m 0644 sqlite3.pc /usr/local/lib/pkgconfig;
Core:sqlite-3.5.4 scoop$ sqlite3 --version
3.5.4
Core:sqlite-3.5.4 scoop$
```

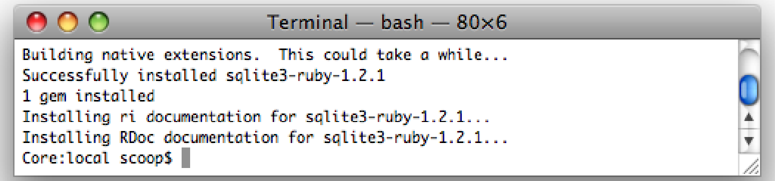
Figure 2.11. Installing SQLite on Mac OS X

Installing the SQLite Database Interface for Ruby

Lastly, we need to install a tiny little module which will allow Ruby to talk to SQLite databases. We'll use the RubyGems system we prepared earlier, which means that the installation boils down to just a single command:

```
$ sudo gem install sqlite3-ruby
```

Running this command should yield the result shown in Figure 2.12.



```
Building native extensions.  This could take a while...
Successfully installed sqlite3-ruby-1.2.1
1 gem installed
Installing ri documentation for sqlite3-ruby-1.2.1...
Installing RDoc documentation for sqlite3-ruby-1.2.1...
Core:local scoop$
```

Figure 2.12. Installing the SQLite database interface for Ruby

Congratulations, Mac users, you're all done!

Installing on Linux

I bet you Linux people smirked when the Mac OS X guys had to use the command line (possibly for the first time), didn't you?

Well, if you're running Linux, I'll assume that you're used to the command line, so I won't feel bad throwing you an archaic series of commands to install all the software you need to be up and running with Rails.

One Size Fits All?

There are literally thousands of different distributions of Linux—more than any other operating system. Each distribution has its own quirks and pitfalls, its own package manager, and different permissions settings, and installations are often tweaked and customized over time. So while I've put every effort into ensuring that these instructions are sound, it would be impossible to offer an absolute guarantee that they'll work on any possible installation of Linux without individual tweaking.

If you do run into any problems installing Rails or its constituents on your machine, I recommend you ask for assistance on the friendly SitePoint Ruby forum.[7] Chances are that someone else has experienced the same problem, and will be happy to help you out.

[7] http://www.sitepoint.com/launch/rubyforum/

Using a Package Manager

As I mentioned, many Linux distributions come with their own package managers, including `apt-get` , `yum`, and `rpm`, among others.

Of course, you're free to use the package manager that's bundled with your Linux distribution to install Ruby, and if you become stuck with the instructions given here for whatever reason, that may be a good option for you.

Rather than attempt to cover all the different package managers available, I'll show you how to install Ruby the manual way.

Prerequisites

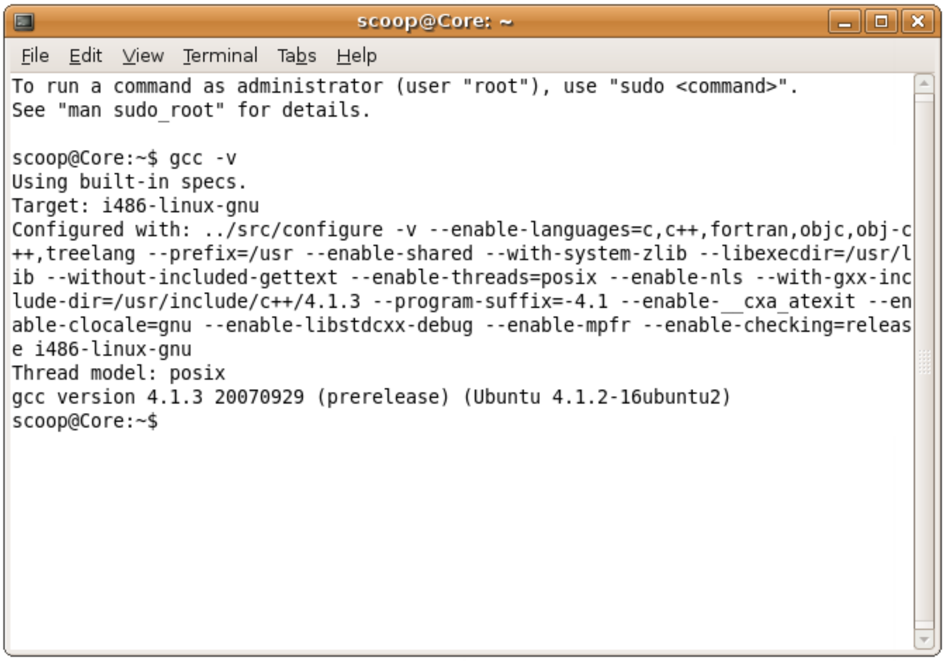
The only prerequisite for installing Ruby on Linux is that you have the `gcc` compiler installed on your machine. `gcc` ships with most Linux distributions by default, but if it's not on your system, you'll either need to use your system's package management system to install it (look for "build essential" or "basic compiler"), or to download a native binary for your system.[8]

Enter the following instructions at the command line to confirm that your compiler is in place:

```
$ gcc -v
```

If the version number for the compiler is displayed, as shown in Figure 2.13, you're ready to install Ruby.

[8] http://gcc.gnu.org/install/binaries.html



```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

scoop@Core:~$ gcc -v
Using built-in specs.
Target: i486-linux-gnu
Configured with: ../src/configure -v --enable-languages=c,c++,fortran,objc,obj-c
++,treelang --prefix=/usr --enable-shared --with-system-zlib --libexecdir=/usr/l
ib --without-included-gettext --enable-threads=posix --enable-nls --with-gxx-inc
lude-dir=/usr/include/c++/4.1.3 --program-suffix=-4.1 --enable-__cxa_atexit --en
able-clocale=gnu --enable-libstdcxx-debug --enable-mpfr --enable-checking=releas
e i486-linux-gnu
Thread model: posix
gcc version 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
scoop@Core:~$
```

Figure 2.13. Confirming the `gcc` compiler is installed

Installing Ruby on Linux

Ruby is available for download from the Ruby ftp site.[9] As mentioned at the outset of this chapter, I recommend the use of version 1.8.6 of the Ruby interpreter.

Build Dependencies

Here's a quick tip if you're using a Debian-based Linux distribution—Ubuntu, for example). Before compiling Ruby, make sure you have all the required packages by entering the following command:

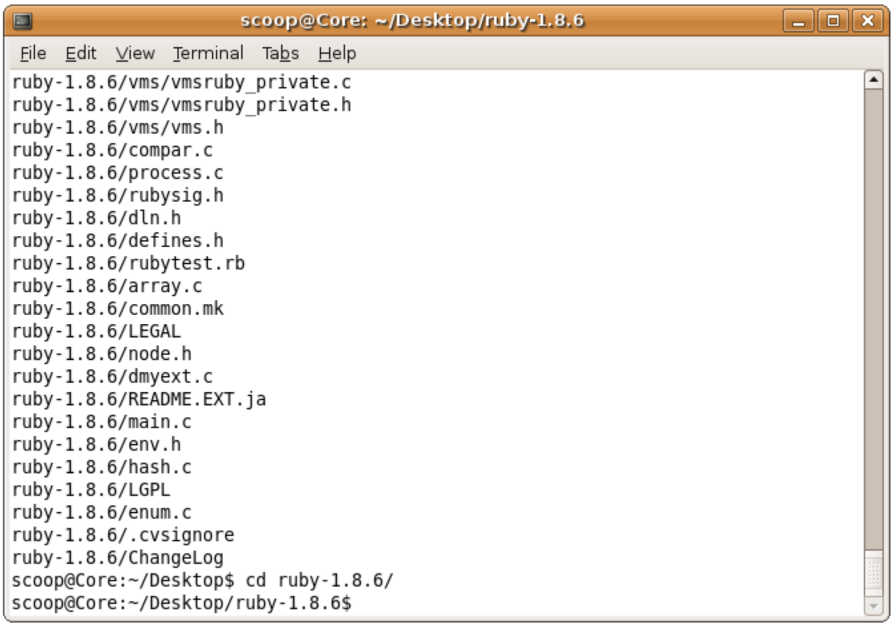
```
$ apt-get build-dep ruby1.8
```

Download the appropriate **tar** file for Ruby (this will be named something like **ruby-1.8.6.tar.gz**), and extract the archive using the `gunzip` and `tar` commands:

[9] ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.6.tar.gz

```
$ gunzip ruby-1.8.6.tar.gz
$ tar xvf ruby-1.8.6.tar
$ cd ruby-1.8.6
```

Then change into the new directory that was created, as illustrated in Figure 2.14.


scoop@Core: ~/Desktop/ruby-1.8.6
File Edit View Terminal Tabs Help

```
ruby-1.8.6/vms/vmsruby_private.c
ruby-1.8.6/vms/vmsruby_private.h
ruby-1.8.6/vms/vms.h
ruby-1.8.6/compar.c
ruby-1.8.6/process.c
ruby-1.8.6/rubysig.h
ruby-1.8.6/dln.h
ruby-1.8.6/defines.h
ruby-1.8.6/rubytest.rb
ruby-1.8.6/array.c
ruby-1.8.6/common.mk
ruby-1.8.6/LEGAL
ruby-1.8.6/node.h
ruby-1.8.6/dmyext.c
ruby-1.8.6/README.EXT.ja
ruby-1.8.6/main.c
ruby-1.8.6/env.h
ruby-1.8.6/hash.c
ruby-1.8.6/LGPL
ruby-1.8.6/enum.c
ruby-1.8.6/.cvsignore
ruby-1.8.6/ChangeLog
scoop@Core:~/Desktop$ cd ruby-1.8.6/
scoop@Core:~/Desktop/ruby-1.8.6$
```

Figure 2.14. Extracting the Ruby archive on Linux

From this directory, run the following command to compile and install Ruby in **/usr/local**:

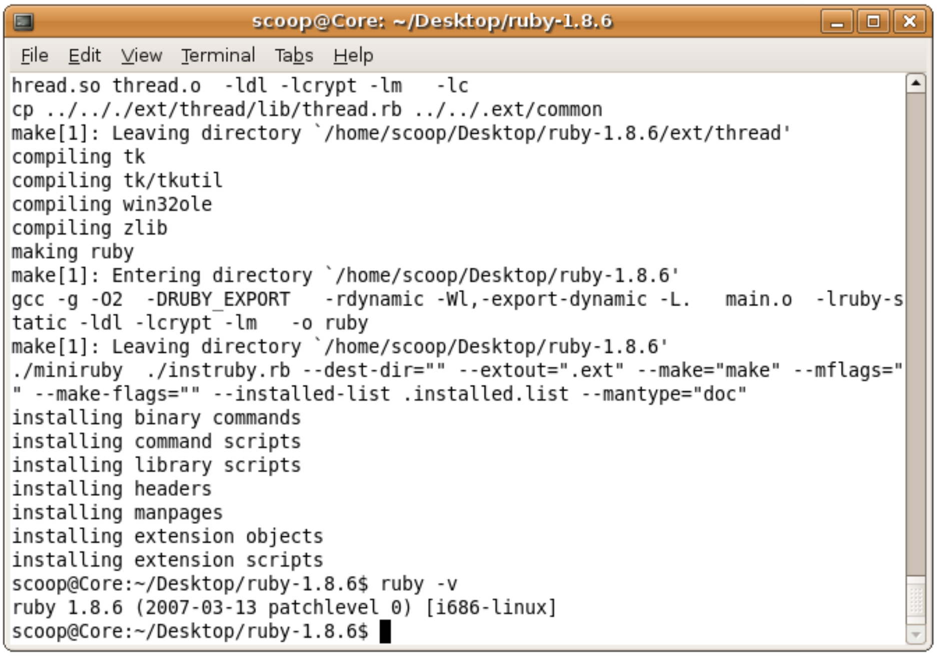
```
$ ./configure && make && sudo make install
```

This process may take 20 minutes or more, so be patient.

Once it's completed, you should add **/usr/local/bin** to your PATH environment variable. I'll assume that, being a Linux user, you know how to do that. Once that environment variable is set, you can enter the following command to check which version of Ruby you installed:

```
$ ruby -v
```

The message displayed should confirm that you're running version 1.8.6, as Figure 2.15 illustrates.


scoop@Core: ~/Desktop/ruby-1.8.6
File Edit View Terminal Tabs Help

```
hread.so thread.o  -ldl -lcrypt -lm   -lc
cp ../.././ext/thread/lib/thread.rb ../.././.ext/common
make[1]: Leaving directory `/home/scoop/Desktop/ruby-1.8.6/ext/thread'
compiling tk
compiling tk/tkutil
compiling win32ole
compiling zlib
making ruby
make[1]: Entering directory `/home/scoop/Desktop/ruby-1.8.6'
gcc -g -O2  -DRUBY_EXPORT   -rdynamic -Wl,-export-dynamic -L.   main.o  -lruby-s
tatic -ldl -lcrypt -lm   -o ruby
make[1]: Leaving directory `/home/scoop/Desktop/ruby-1.8.6'
./miniruby  ./instruby.rb --dest-dir="" --extout=".ext" --make="make" --mflags="
" --make-flags="" --installed-list .installed.list --mantype="doc"
installing binary commands
installing command scripts
installing library scripts
installing headers
installing manpages
installing extension objects
installing extension scripts
scoop@Core:~/Desktop/ruby-1.8.6$ ruby -v
ruby 1.8.6 (2007-03-13 patchlevel 0) [i686-linux]
scoop@Core:~/Desktop/ruby-1.8.6$
```

Figure 2.15. Installing Ruby on Linux

Now, on to the next step: installing RubyGems.

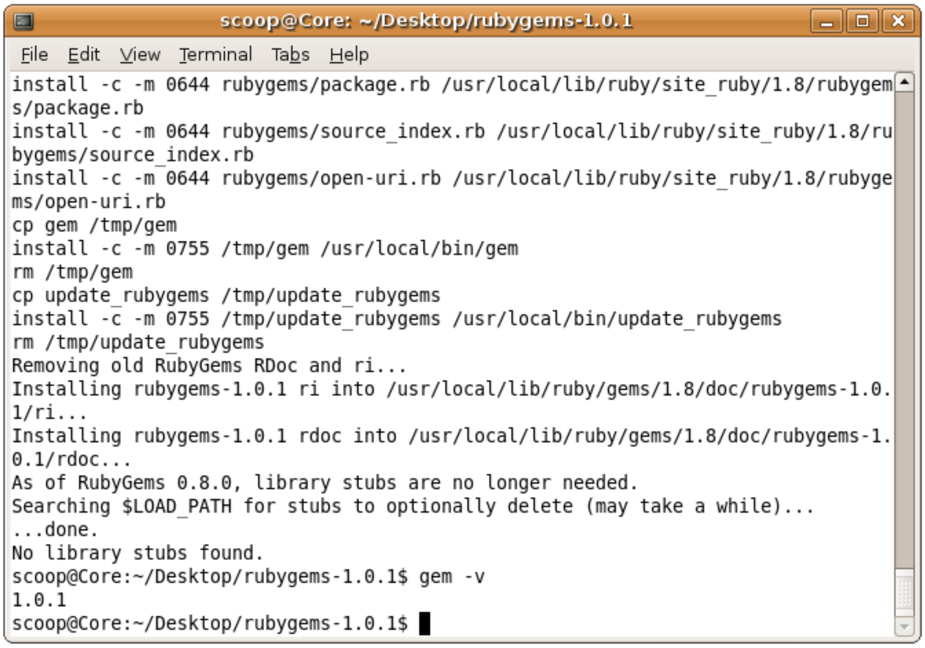
Installing RubyGems on Linux

Next up is the installation of RubyGems, the package manager for Ruby-related software. RubyGems works much like the package manager that your operating system uses to manage the various Linux utilities installed on your machine. RubyGems makes it easy to install all sorts of additional software and extensions for Ruby.

RubyGems is available for download from http://rubyforge.org/projects/rubygems/. Once you've downloaded and extracted it, change to the **rubygems** directory and run the following command:

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

This will set up and install RubyGems for use on your system, and also make the `gem` command available for you to use—the `gem` command is what we'll use to install Rails itself. It shouldn't take long, and once it completes, you can execute the `gem` command to confirm that your installation was successful. The output should look like that shown in Figure 2.16.


scoop@Core: ~/Desktop/rubygems-1.0.1
File Edit View Terminal Tabs Help

```
install -c -m 0644 rubygems/package.rb /usr/local/lib/ruby/site_ruby/1.8/rubygem
s/package.rb
install -c -m 0644 rubygems/source_index.rb /usr/local/lib/ruby/site_ruby/1.8/ru
bygems/source_index.rb
install -c -m 0644 rubygems/open-uri.rb /usr/local/lib/ruby/site_ruby/1.8/rubyge
ms/open-uri.rb
cp gem /tmp/gem
install -c -m 0755 /tmp/gem /usr/local/bin/gem
rm /tmp/gem
cp update_rubygems /tmp/update_rubygems
install -c -m 0755 /tmp/update_rubygems /usr/local/bin/update_rubygems
rm /tmp/update_rubygems
Removing old RubyGems RDoc and ri...
Installing rubygems-1.0.1 ri into /usr/local/lib/ruby/gems/1.8/doc/rubygems-1.0.
1/ri...
Installing rubygems-1.0.1 rdoc into /usr/local/lib/ruby/gems/1.8/doc/rubygems-1.
0.1/rdoc...
As of RubyGems 0.8.0, library stubs are no longer needed.
Searching $LOAD_PATH for stubs to optionally delete (may take a while)...
...done.
No library stubs found.
scoop@Core:~/Desktop/rubygems-1.0.1$ gem -v
1.0.1
scoop@Core:~/Desktop/rubygems-1.0.1$
```

Figure 2.16. Installing RubyGems on Linux

We have successfully installed RubyGems. Now we can finally install the Rails framework!

Updating RubyGems

RubyGems is constantly being developed and improved, and new versions are released frequently. If you ever need to update RubyGems—for compatibility reasons, for example—simply enter the following at the command line:

```
$ sudo gem update --system
```

Installing Rails on Linux

Using RubyGems, the installation of Rails itself is a breeze. To install Rails, type the following input at the command prompt as the `root` user (or using `sudo` if it's installed on your system):

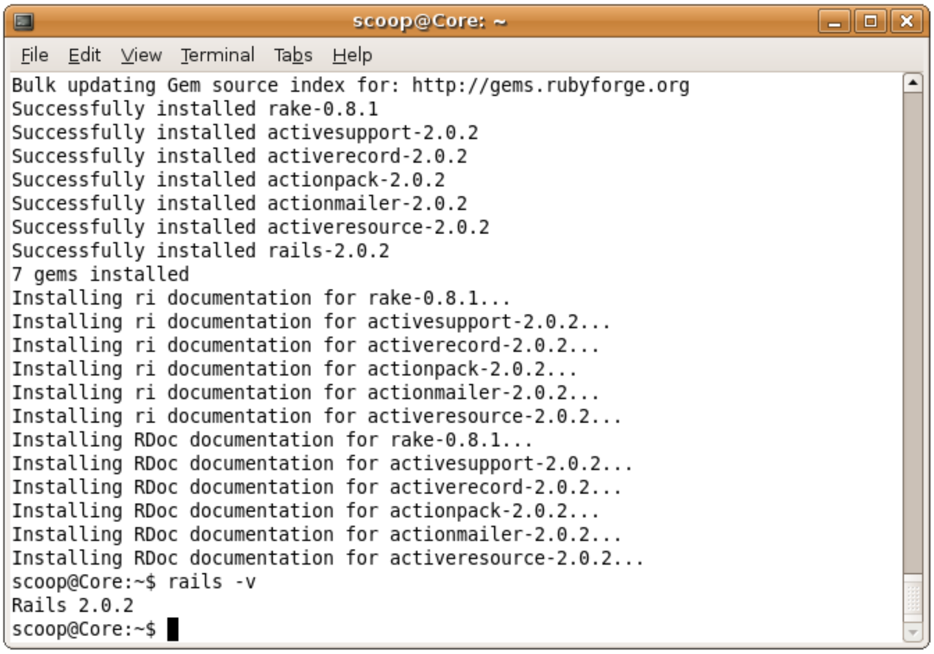
```
$ sudo gem install rails
```

The process may take ten minutes or so, depending on the speed of your Internet connection, but that's all you need to do! And as an added bonus, RubyGems gives us an easy way to stay up to date with future Rails releases—whenever we want to upgrade Rails, we just need to type this command!

To confirm that your Rails installation was successful, type the following command to display the version of Rails that was installed:

```
$ rails -v
```

The result that you see should be the same as that shown in Figure 2.17.



```
Bulk updating Gem source index for: http://gems.rubyforge.org
Successfully installed rake-0.8.1
Successfully installed activesupport-2.0.2
Successfully installed activerecord-2.0.2
Successfully installed actionpack-2.0.2
Successfully installed actionmailer-2.0.2
Successfully installed activeresource-2.0.2
Successfully installed rails-2.0.2
7 gems installed
Installing ri documentation for rake-0.8.1...
Installing ri documentation for activesupport-2.0.2...
Installing ri documentation for activerecord-2.0.2...
Installing ri documentation for actionpack-2.0.2...
Installing ri documentation for actionmailer-2.0.2...
Installing ri documentation for activeresource-2.0.2...
Installing RDoc documentation for rake-0.8.1...
Installing RDoc documentation for activesupport-2.0.2...
Installing RDoc documentation for activerecord-2.0.2...
Installing RDoc documentation for actionpack-2.0.2...
Installing RDoc documentation for actionmailer-2.0.2...
Installing RDoc documentation for activeresource-2.0.2...
scoop@Core:~$ rails -v
Rails 2.0.2
scoop@Core:~$ 
```

Figure 2.17. Installing Rails via RubyGems on Linux

All that's left now is to install a database—then we can get to work!

Installing SQLite on Linux

Most modern Linux distributions may or may not come packaged with a (more or less) recent version of SQLite and you're free to use that. It is crucial, however, that you're installing SQLite 3.x (as opposed to SQLite 2.x).

In case your Linux distribution doesn't ship with a prepackaged version of SQLite, follow the simple installation instructions found below.

SQLite is available for download from http://www.sqlite.org/download.html. The rest of these instructions assume you download the source tarball. As of this writing, the most recent version of SQLite available was 3.5.4.

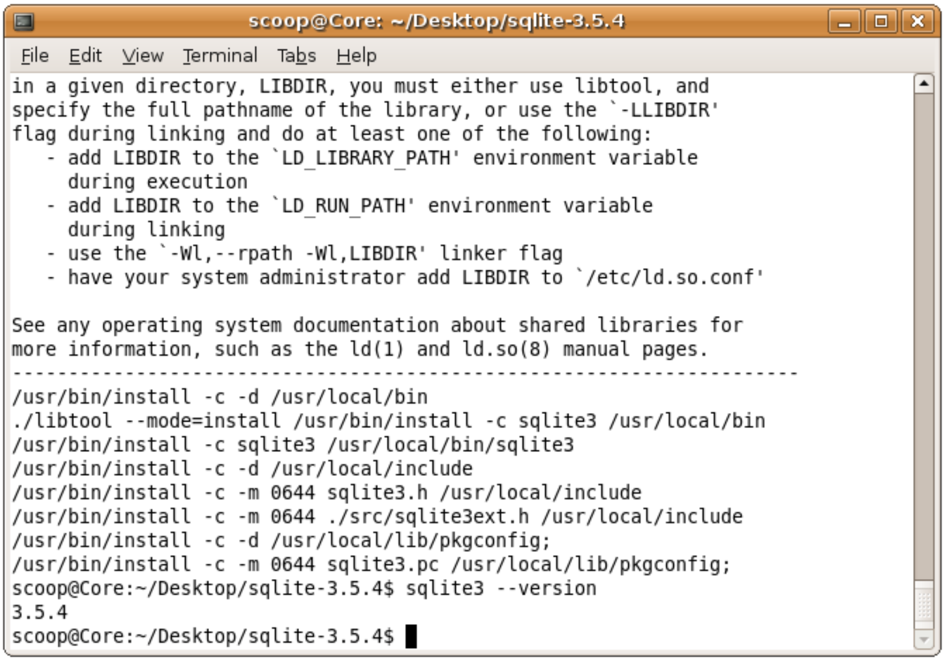
Once you have the file, it's time to extract and compile it using the following batch of commands:

```
$ tar zxvf sqlite-3.5.4.tar.gz
$ cd sqlite-3.5.4
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

At this point you have successfully installed SQLite on your Linux system. To confirm that, the following command will print out the version of SQLite that you downloaded and installed:

```
$ sqlite3 --version
```

Assuming that you have the directory `/usr/local/bin` in your operating system `PATH`, Figure 2.18 shows the desired output.


scoop@Core: ~/Desktop/sqlite-3.5.4
File Edit View Terminal Tabs Help

```
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,--rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
/usr/bin/install -c -d /usr/local/bin
./libtool --mode=install /usr/bin/install -c sqlite3 /usr/local/bin
/usr/bin/install -c sqlite3 /usr/local/bin/sqlite3
/usr/bin/install -c -d /usr/local/include
/usr/bin/install -c -m 0644 sqlite3.h /usr/local/include
/usr/bin/install -c -m 0644 ./src/sqlite3ext.h /usr/local/include
/usr/bin/install -c -d /usr/local/lib/pkgconfig;
/usr/bin/install -c -m 0644 sqlite3.pc /usr/local/lib/pkgconfig;
scoop@Core:~/Desktop/sqlite-3.5.4$ sqlite3 --version
3.5.4
scoop@Core:~/Desktop/sqlite-3.5.4$ 
```

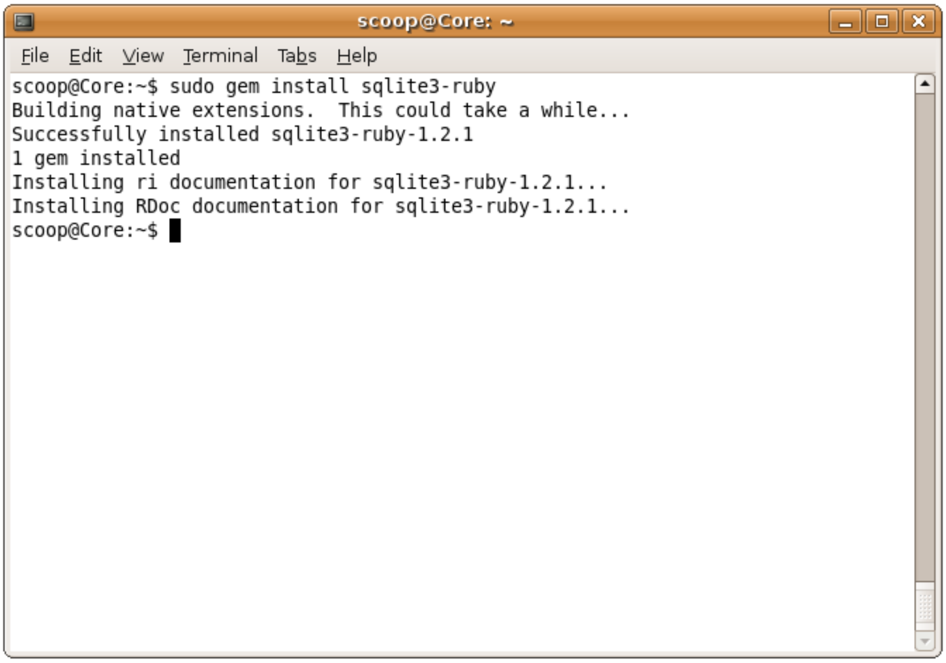
Figure 2.18. Confirming the successful installation of SQLite

Installing the SQLite Database Interface for Ruby

Lastly, we need to install a tiny little module that allows Ruby to talk to SQLite databases. To do so, we'll use the RubyGems system we've installed earlier. Because of that, the installation boils down to a single command only:

```
$ sudo gem install sqlite3-ruby
```

Figure 2.19 shows the desired result.



```
scoop@Core:~$ sudo gem install sqlite3-ruby
Building native extensions.  This could take a while...
Successfully installed sqlite3-ruby-1.2.1
1 gem installed
Installing ri documentation for sqlite3-ruby-1.2.1...
Installing RDoc documentation for sqlite3-ruby-1.2.1...
scoop@Core:~$
```

Figure 2.19. Installing the SQLite Ruby interface

Congratulations, now you're all done!

Building a Foundation

Is everyone prepared? Good! Now that you've put your workstation on Rails, let's do something with it. In this section, we'll build the foundations of the application that we'll develop throughout the rest of this book.

One Directory Structure to Rule Them All

In Chapter 1, I mentioned that Rails applications follow certain conventions. One of these conventions is that an application written in Rails always has the same directory structure—one in which every file has its designated place. By gently forcing this directory structure upon developers, Rails ensures that your work is semi-automatically organized the Rails way.

Figure 2.20 shows what the structure looks like. We'll create this directory structure for our application in just a moment.

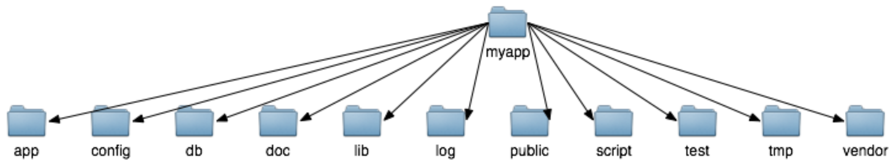


Figure 2.20. The default directory structure for a Rails application

As you can see, this standard directory structure consists of quite a few subdirectories (and I'm not even showing *their* subdirectories yet!). This wealth of subdirectories can be overwhelming at first, but we'll explore them one by one. A lot of thought has gone into establishing and naming the folders, and the result is an application with a well structured file system.

Before you go and manually create all these directories yourself, let me show you how to set up that pretty directory structure using just one command—I told you that Rails allows us to do *less* typing!

Creating the Standard Directory Structure

It's easy to generate the default directory structure shown in Figure 2.20 for a new Rails application using the `rails` command.

Before we start, I'd like to introduce you to the secret, under-the-hood project name we'll give to our digg clone: *Shovell*. Yes, it's cheeky, but it'll work.

Now, let's go ahead and create the directory structure to hold our application.

A Regular Console Window Just Won't Do!

If you're a Windows user, you might be tempted to fire up a regular DOS console for your command line work. This won't work at all, I'm afraid.

Instead, launch a Ruby console by starting Instant Rails, then clicking on the `I` button at the top-left corner of the control panel. From the menu that appears, select **Rails Applications** > **Open Ruby Console Window**, as pictured in Figure 2.21.

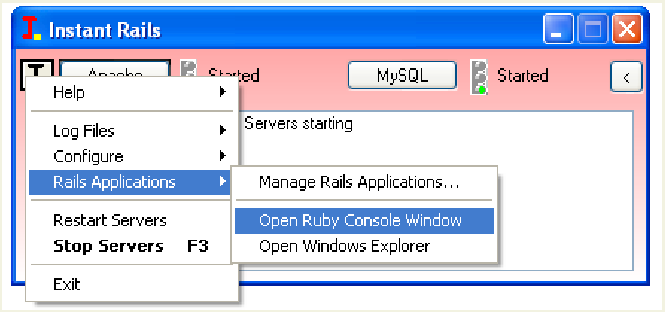
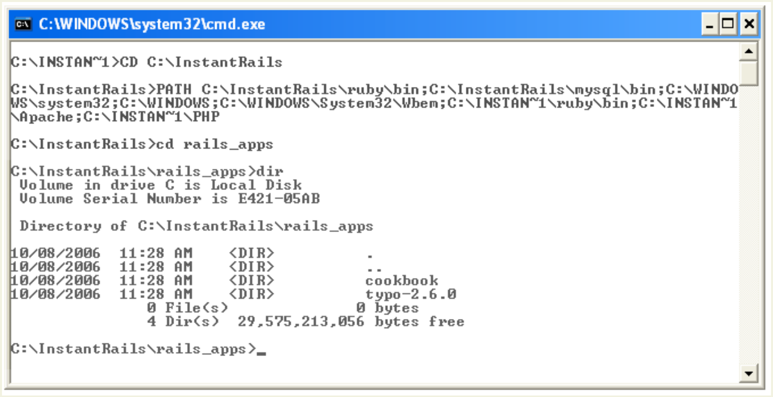


Figure 2.21. Launching a console window from Instant Rails

The Ruby console must be used, because Instant Rails doesn't modify anything in your regular Windows environment when it installs. Launching a console from the Instant Rails control panel ensures that your console will be loaded with all the environment settings that Rails needs. The Windows Ruby console is depicted in Figure 2.22.



```
C:\INSTAN~1>CD C:\InstantRails

C:\InstantRails>PATH C:\InstantRails\ruby\bin;C:\InstantRails\mysql\bin;C:\WINDO
WS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\INSTAN~1\ruby\bin;C:\INSTAN~1
\Apache;C:\INSTAN~1\PHP

C:\InstantRails>cd rails_apps

C:\InstantRails\rails_apps>dir
 Volume in drive C is Local Disk
 Volume Serial Number is E421-05AB

 Directory of C:\InstantRails\rails_apps

10/08/2006  11:28 AM    <DIR>          .
10/08/2006  11:28 AM    <DIR>          ..
10/08/2006  11:28 AM    <DIR>          cookbook
10/08/2006  11:28 AM    <DIR>          typo-2.6.0
               0 File(s)              0 bytes
               4 Dir(s)  29,575,213,056 bytes free

C:\InstantRails\rails_apps>_
```

Figure 2.22. The Ruby Console under Windows

The `rails` command takes a single parameter: the directory where you'd like to store your application. You can, and are encouraged to, execute it from the parent directory in which you want your new Rails application to live. I'll do this right in my home directory:

```
$ rails shovell
create
create app/controllers
create app/helpers
create app/models
create app/views/layouts
create config/environments
create config/initializers
create db
create doc
create lib
create lib/tasks
create log
⋮ log entries…
```

Congratulations, your directory structure has been created!

Starting Our Application

Even before we write any code, it's possible to start up our application environment to check that our setup is working correctly. This exercise should give us a nice boost of confidence before we progress any further.

What we'll do is launch **Mongrel**, a fast, stand-alone HTTP library and web server for Ruby. Mongrel is included with the Ruby installation that we stepped through earlier in this chapter, so it's installed on our machine and ready to use.

In previous versions of Rails the default built-in web server was WEBrick. WEBrick is still included with Rails and can be used instead of Mongrel if desired (the Mongrel output below tells you how). For our purposes, however, Mongrel will do just fine.

WEBrick or Mongrel?

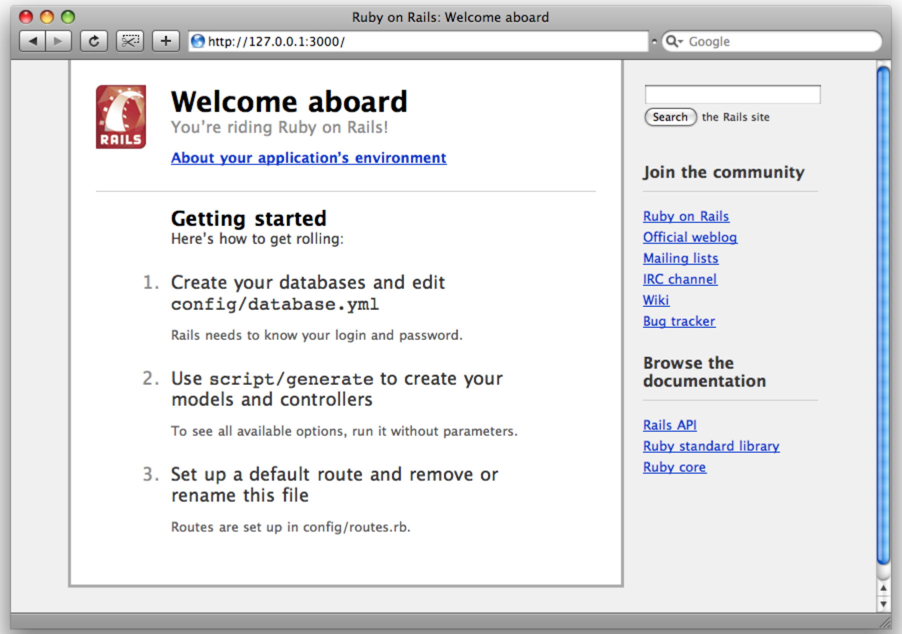
If you installed Rails on Mac OS X 10.4 or Linux, you may find WEBrick is still your default web server. This is fine; there's no pressing reason to use Mongrel. If you'd like to install it anyway, you can do so via the command: `sudo gem install mongrel`.

To start up Mongrel—and the rest of the Rails environment for our application—we return once again to the command line. Change into the **Shovell** subdirectory that was created when we executed the `rails` command in the previous section. From the **shovell** directory, enter the command `ruby script/server`.

This command will fire up the Mongrel web server, which will then begin to listen for requests on TCP port 3000 of your local machine:

```
$ cd shovell
$ ruby script/server
ruby script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
** Ruby version is up-to-date; cgi_multipart_eof_fix was not loaded
** Starting Mongrel listening at 0.0.0.0:3000
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready.  TERM => stop.  USR2 => restart.  INT => stop (no
 restart).
** Rails signals registered.  HUP => reload (without restart).  It
might not work well.
** Mongrel available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Well done: you just started up your application for the first time! Okay, so it's not going to be doing a whole lot—we haven't written any lines of code yet, after all—but you can now connect to your application by entering http://localhost:3000/ into your web browser's address bar; you should see something similar to Figure 2.23.


Ruby on Rails: Welcome aboard

http://127.0.0.1:3000/

Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Getting started

Here's how to get rolling:

1. Create your databases and edit `config/database.yml`

   Rails needs to know your login and password.

2. Use `script/generate` to create your models and controllers

   To see all available options, run it without parameters.

3. Set up a default route and remove or rename this file

   Routes are set up in config/routes.rb.

Search the Rails site

Join the community

Ruby on Rails
Official weblog
Mailing lists
IRC channel
Wiki
Bug tracker

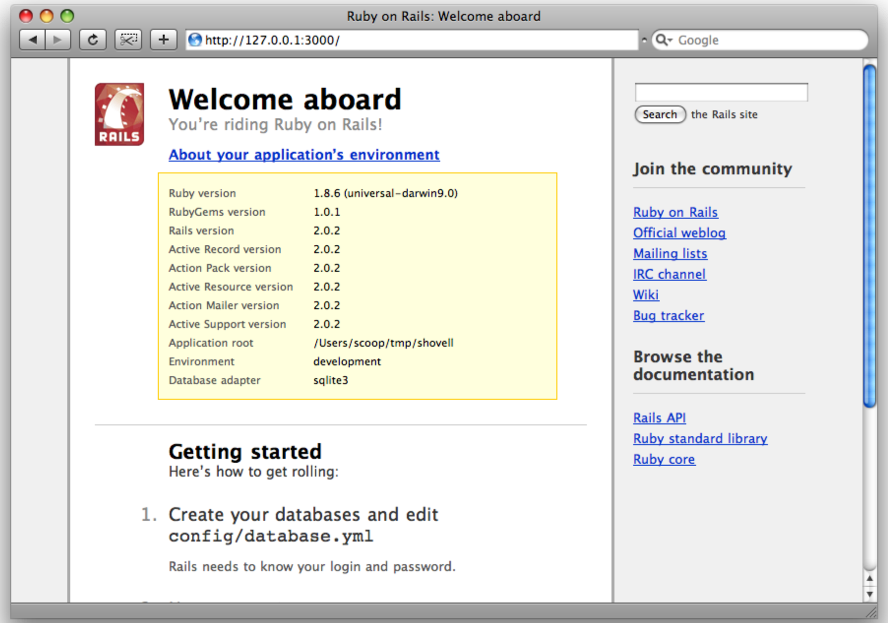
Browse the documentation

Rails API
Ruby standard library
Ruby core

Figure 2.23. Welcome aboard: the Rails welcome screen

This welcome screen provides us with quite a few items, including some steps for getting started with Rails application development. Don't investigate these just yet; we'll deal with that soon enough. You'll also notice in the sidebar some links to sites such as the Rails wiki and the mailing list archives, which you can browse through at your leisure. And there are some links to documentation for Rails and for Ruby; you'll find these resources useful once you've progressed further with Rails development.

If you're interested to see the version numbers of each of the components we've installed, click on the link labeled **About your application's environment**. You'll see a nicely animated information box, like the one in Figure 2.24. This dialog contains all the version information you'll ever likely to need. If you've followed the installation instructions in this book, you should have the latest versions of everything.



Welcome aboard
You're riding Ruby on Rails!

About your application's environment

Ruby version	1.8.6 (universal-darwin9.0)
RubyGems version	1.0.1
Rails version	2.0.2
Active Record version	2.0.2
Action Pack version	2.0.2
Active Resource version	2.0.2
Action Mailer version	2.0.2
Active Support version	2.0.2
Application root	/Users/scoop/tmp/shovell
Environment	development
Database adapter	sqlite3

Getting started
Here's how to get rolling:

1. Create your databases and edit `config/database.yml`

   Rails needs to know your login and password.

Join the community

Ruby on Rails
Official weblog
Mailing lists
IRC channel
Wiki
Bug tracker

Browse the documentation

Rails API
Ruby standard library
Ruby core

Figure 2.24. Viewing version information

Okay, so you're finally ready to write some code. But wait! Which text editor will you be using?

Which Text Editor?

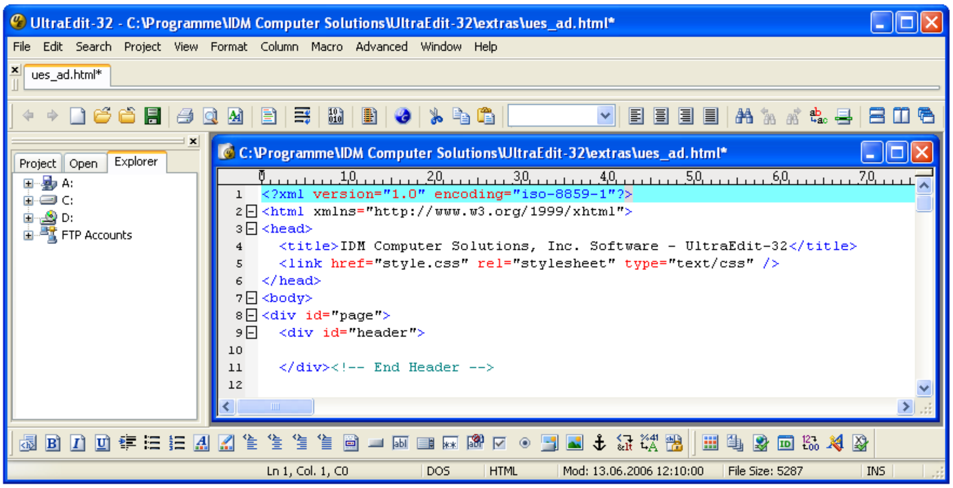
The question of which text editor is best for web development has spawned arguments that border on religious fanaticism. However, while it's certainly possible to develop Rails applications using the default text editor that comes bundled with your operating system, I don't recommend it—the benefits provided by a specifically designed programmer's editor can prevent typing errors and increase your productivity immeasurably.

In this section, I've suggested a couple of alternatives for each operating system, and I'll let you make a choice that suits your personal preferences and budget.

Windows Text Editors

UltraEdit

The most popular option for editing Rails code on Windows seems to be UltraEdit, which is shown in Figure 2.25. UltraEdit is available for download as a free trial, and may be purchased online for US$49.95.[10] It offers syntax highlighting, code completion, and proper Unicode support (for international characters), as well as providing the facility to jump quickly between several files. This last is a huge plus for Rails applications, which usually consist of several dozen files!


UltraEdit-32 - C:\Programme\IDM Computer Solutions\UltraEdit-32\extras\ues_ad.html*
File Edit Search Project View Format Column Macro Advanced Window Help
ues_ad.html*
Project Open Explorer
A:
C:
D:
FTP Accounts
C:\Programme\IDM Computer Solutions\UltraEdit-32\extras\ues_ad.html*
Ln 1, Col. 1, C0 DOS HTML Mod: 13.06.2006 12:10:00 File Size: 5287 INS

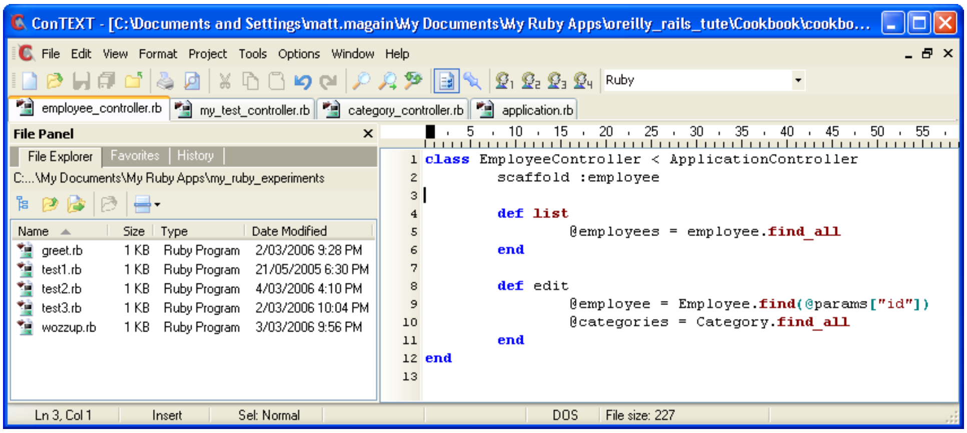
```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>IDM Computer Solutions, Inc. Software - UltraEdit-32</title>
  <link href="style.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div id="page">
  <div id="header">

  </div><!-- End Header -->

```

Figure 2.25. UltraEdit: a powerful Windows editor

ConTEXT

A free text editor alternative is ConTEXT,[11] shown in Figure 2.26, which holds its own in the features department. ConTEXT also supports syntax highlighting for Ruby (which is available as a separate download),[12] the ability to open multiple documents in tabs, and a host of other features to make your development experience more enjoyable. I especially like the fact that ConTEXT is quite lightweight, so it loads very quickly. Oh, and that lack of a price tag is rather attractive, too!

[10] http://www.ultraedit.com/
[11] http://context.cx/
[12] http://www.context.cx/component/option,com_docman/task,cat_view/gid,76/Itemid,48/


ConTEXT - [C:\Documents and Settings\matt.magain\My Documents\My Ruby Apps\oreilly_rails_tute\Cookbook\cookbo...
File Edit View Format Project Tools Options Window Help
Ruby
employee_controller.rb
my_test_controller.rb
category_controller.rb
application.rb
File Panel
File Explorer
Favorites
History
C:...\My Documents\My Ruby Apps\my_ruby_experiments
Ln 3, Col 1
Insert
Sel: Normal
DOS
File size: 227

Name	Size	Type	Date Modified
greet.rb	1 KB	Ruby Program	2/03/2006 9:28 PM
test1.rb	1 KB	Ruby Program	21/05/2005 6:30 PM
test2.rb	1 KB	Ruby Program	4/03/2006 4:10 PM
test3.rb	1 KB	Ruby Program	2/03/2006 10:04 PM
wozzup.rb	1 KB	Ruby Program	3/03/2006 9:56 PM

```
class EmployeeController < ApplicationController
        scaffold :employee

        def list
                @employees = employee.find_all
        end

        def edit
                @employee = Employee.find(@params["id"])
                @categories = Category.find_all
        end
end

```

Figure 2.26. ConTEXT: a free, feature-rich text editor for Windows

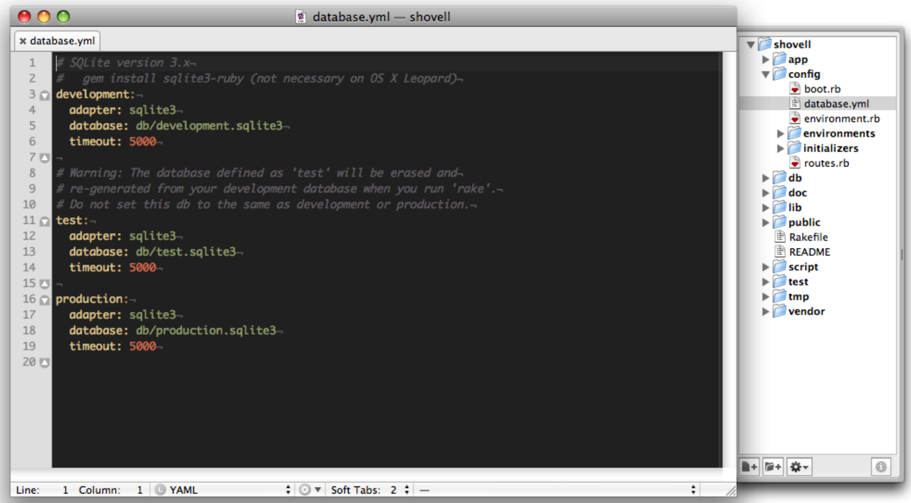
Mac OS X Text Editors

TextMate

For Mac OS X users, the hands-down winner in the Rails code editor popularity contest is TextMate,[13] which is shown in Figure 2.27. TextMate is the editor you can see in action in numerous screencasts[14] available from the Rails web site. It's available for download as a free, 30-day trial, and the full version of the product costs €39.

TextMate boasts terrific project management support, amazing macro recording/code completion functionality, and one of the most complete syntax highlighting implementations for Rails code. As you can probably tell, I'm a big fan, and recommend it heartily. But this is beginning to sound like a television commercial, so I'll leave it at that.

[13] http://www.macromates.com/
[14] http://www.rubyonrails.org/screencasts/



```
# SQLite version 3.x
#   gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  timeout: 5000

# 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:
  adapter: sqlite3
  database: db/test.sqlite3
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  timeout: 5000
```

Line: 1 Column: 1 YAML Soft Tabs: 2

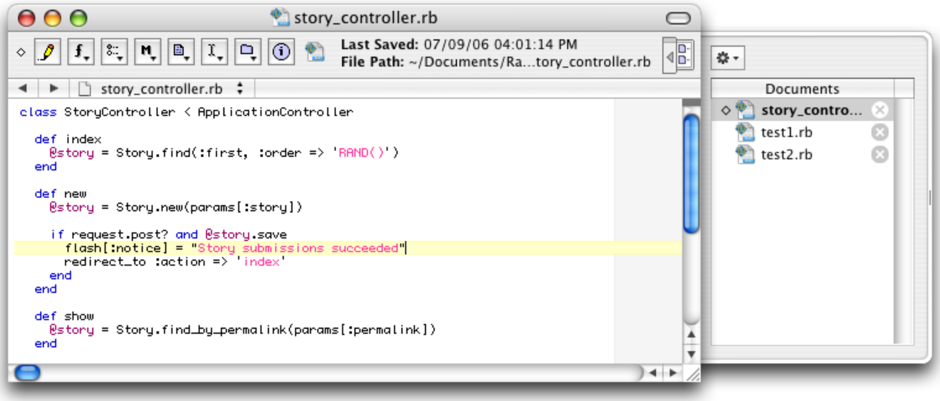
Figure 2.27. TextMate running under Mac OS X

TextWrangler

TextWrangler is a free, simple text editor made by BareBones Software. As with the other editors listed here, TextWrangler tidies up your workspace by allowing you to have several files open at the same time. The documents are listed in a pull-out "drawer" to one side of the interface, rather than as tabs.

You can download TextWrangler from the BareBones Software web site.[15] Figure 2.28 shows TextWrangler in action.

[15] http://barebones.com/products/textwrangler/


story_controller.rb
Last Saved: 07/09/06 04:01:14 PM
File Path: ~/Documents/Ra...tory_controller.rb
story_controller.rb
Documents
story_contro...
test1.rb
test2.rb

```
class StoryController < ApplicationController

  def index
    @story = Story.find(:first, :order => 'RAND()')
  end

  def new
    @story = Story.new(params[:story])

    if request.post? and @story.save
      flash[:notice] = "Story submissions succeeded"
      redirect_to :action => 'index'
    end
  end

  def show
    @story = Story.find_by_permalink(params[:permalink])
  end
```

Figure 2.28. TextWrangler, a free text editor for Mac OS X

Linux and Cross-platform Editors

A number of development-centric text editors that run on a variety of platforms are available for free download. The following editors have loyal followings, and all run equally well on Linux as on Microsoft Windows and Mac OS X:

- Emacs, http://www.emacswiki.org/
- jEdit, http://www.jedit.org/
- RadRails, http://www.radrails.org/
- Vim, http://www.vim.org/

A more comprehensive list of text editors can be found in the Rails Wiki.[16] This page also covers potential enhancement modules for other editors.

Summary

In this chapter, I showed you how to install all the software you need to develop a web application in Ruby on Rails.

We installed Ruby, Rails, and SQLite, and set up the standard directory structure for our application, which we've named "Shovell." We even launched the application for the first time, which enabled us to check which versions we were running of the components involved. And finally, I gave you some options for text editors you can use to build the application.

[16] http://wiki.rubyonrails.org/rails/pages/Editors/

All this work has been completed in preparation for Chapter 4, where we'll begin to write our first lines of application code. But first, there's some theory we have to tackle. Hold on tight, we'll start coding soon enough!

Chapter 3

Introducing Ruby

While this chapter certainly makes no attempt to constitute a complete guide to the Ruby language, it will introduce you to some of the basics of Ruby. We'll power through a crash course in object oriented programming that covers the more common features of the language, and leave the more obscure aspects of Ruby for a dedicated reference guide.[1] I'll also point out some of the advantages that Ruby has over other languages when it comes to developing applications for the Web.

Some Rails developers suggest that it's possible to learn and use Rails without learning the Ruby basics first, but as far as I'm concerned, it's extremely beneficial to know even a *little* Ruby before diving into the guts of Rails. In fact, you'll automatically become a better Rails programmer.

Ruby Is a Scripting Language

In general, programming languages fall into one of two categories: they're either compiled languages or scripting languages. Let's explore what each of those terms means, and understand the differences between them.

[1] http://www.ruby-doc.org/stdlib/

Compiled Languages

The language in which you write an application is not actually something that your computer understands. Your code needs to be translated into bits and bytes that can be executed by your computer. This process of translation is called **compilation**, and any language that requires compilation is referred to as a **compiled language**. Examples of compiled languages include C, C#, and Java.

For a compiled language, the actual compilation is the final step in the development process. You invoke a **compiler**—the software program that translates your final handwritten, human-readable code into machine-readable code—and the compiler creates an executable file. This final product is then able to execute independently of the original source code.

Thus, if you make changes to your code, and you want those changes to be incorporated into the application, you must stop the running application, recompile it, then start the application again.

Scripting Languages

On the other hand, a scripting language such as Ruby, PHP, or Python relies upon an application's source code all the time. **Scripting languages** don't have a compiler or a compilation phase per se; instead, they use an **interpreter**—a program that runs on the web server—to translate handwritten code into machine-executable code on the fly. The link between the running application and your handcrafted code is never severed, because that scripting code is translated every time it is invoked; in other words, for every web page that your application renders.

As you might have gathered from the name, the use of an interpreter rather than a compiler is the major difference between a scripting language and a compiled language.

The Great Performance Debate

If you've come from a compiled-language background, you might be concerned by all this talk of translating code on the fly—how does it affect the application's performance?

These concerns are valid. Translating code on the web server every time it's needed is certainly more expensive, performance-wise, than executing precompiled code, as it requires more effort on the part of your machine's processor. The good news is that there are ways to speed up scripted languages, including techniques such as code caching—caching the output of a script for reuse rather than executing the script every time—and persistent interpreters—loading the interpreter once and keeping it running instead of having to load it for every request. However, performance topics are beyond the scope of this book.

There's also an upside to scripted languages in terms of performance—namely, *your* performance while developing an application.

Imagine that you've just compiled a shiny new Java application, and launched it for the first time … and then you notice an embarrassing typo on the welcome screen. To fix it, you have to stop your application, go back to the source code, fix the typo, wait for the code to recompile, and restart your application to confirm that it is fixed. And if you find another typo, you'll need to repeat that process *again*. Lather, rinse, repeat.

In a scripting language, you can fix the typo and just reload the page in your browser—no restart, no recompile, no nothing. It's as simple as that.

Ruby Is an Object Oriented Language

Ruby, from its very beginnings, was built as a programming language that adheres to the principles of **object oriented programming** (OOP). Before getting into Ruby specifics, let's unpack some fundamental concepts of OOP. The theory can be a bit dry when you're itching to start coding, but we'll cover a lot of ground in this short section. It will hold you in good stead, so don't skip it.

OOP is a programming paradigm that first surfaced in the 1960s, but didn't gain traction until the 1980s with C++. Its core idea is that programs should be composed of individual entities, or **objects**, each of which has the ability to communicate with other objects around it. Additionally, each object may have the facility to store data internally, as depicted in Figure 3.1.

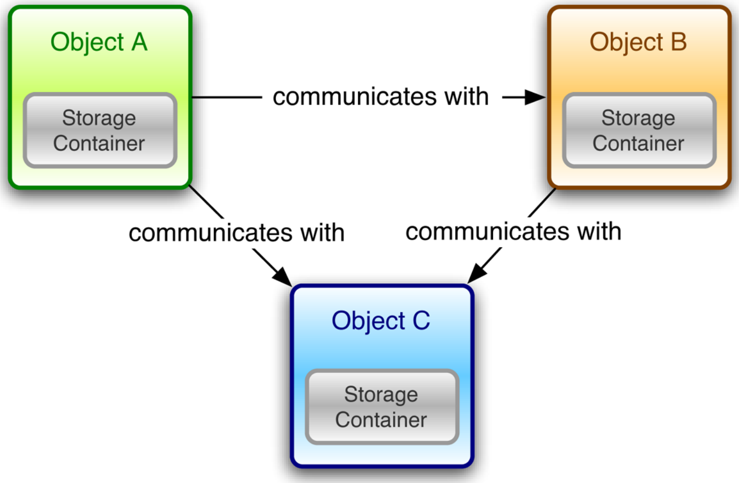


Figure 3.1. Communication between objects

Objects in an OOP application are often modeled on real-world objects, so even non-programmers can usually recognize the basic role that an object plays.

And, just like the real world, OOP defines objects with similar characteristics as belonging to the same **class**. A class is a construct for defining properties for objects that are alike, and equipping them with functionality. For example, a class named `Car` might define the attributes `color` and `mileage` for its objects, and assign them functionality: actions such as `open the trunk`, `start the engine`, and `change gears`. These different actions are known as **methods**, although you'll often see Rails enthusiasts refer to the methods of a controller (a kind of object used in Rails, which you'll become very familiar with) as "actions"; you can safely consider the two terms to be interchangeable.

Understanding the relationship between a class and its objects is integral to understanding how OOP works. For instance, one object can invoke functionality on another object, and can do so without affecting other objects of the same class. So, if one car object was instructed to open its trunk, its trunk would open, but the trunk of other cars would remain closed—think of KITT, the talking car from the television show *Knight Rider*, if it helps with the metaphor.[2] Similarly, if our high-tech talking car were instructed to change color to red, it would do so, but other cars would not.

[2] *Knight Rider* [http://en.wikipedia.org/wiki/Knight_Rider] was a popular 1980s series which featured modern-day cowboy Michael Knight (played by David Hasselhoff) and his opinionated, talking, black

When we create a new object in OOP, we base it on an existing class. The process of creating new objects from a class is called **instantiation**. Figure 3.2 illustrates this concept.

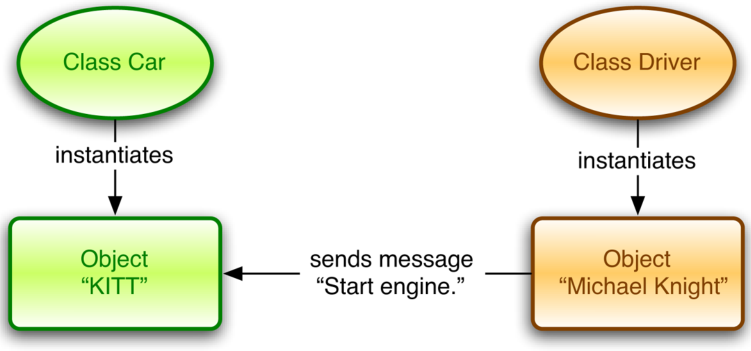


Figure 3.2. Classes and objects

As I mentioned, objects can communicate with each other and invoke functionality (methods) on other objects. Invoking an object's methods can be thought of as asking the object a question, and getting an answer in return.

Consider the example of our famous talking car again. Let's say we ask the talking car object to report its current mileage. This question is not ambiguous: the answer that the object gives is called a **return value**, and is shown in Figure 3.3.

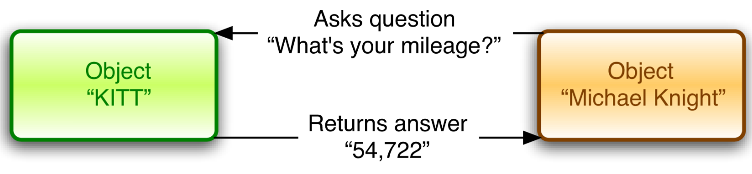


Figure 3.3. Asking a simple question

In some cases, the question-and-answer analogy doesn't quite fit. In these situations, we might rephrase the analogy to consider the question to be an instruction, and the answer a status report indicating whether or not the instruction was executed successfully. This process might look something like the diagram in Figure 3.4.

Pontiac Firebird named KITT. If you missed it in the '80s, you may be more familiar with the Val Kilmer-voiced Ford Mustang in the 2008 remake. Don't worry, having seen the show isn't a prerequisite to understanding object oriented programming!

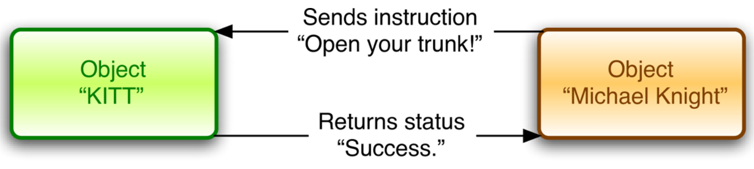


Figure 3.4. Sending instructions

Sometimes we need a bit more flexibility with our instructions. For example, if we wanted to tell our car to change gear, we need to tell it not only to change gear, but also which gear to change to. The process of asking these kinds of questions is referred to as passing an argument to the method.

An **argument** is an input value that's provided to a method. An argument can be used in two ways:

- to influence how a method operates
- to influence which object a method operates on

An example is shown in Figure 3.5, where the method is "change gear," and the number of the gear to which the car must change (two) is the argument.

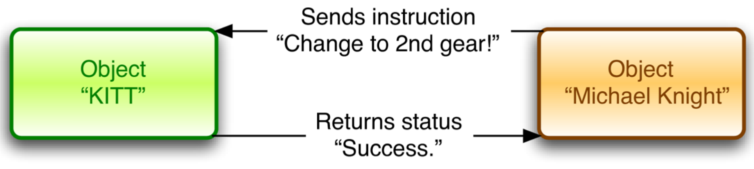


Figure 3.5. Passing arguments

A more general view of all of these different types of communication between objects is this: invoking an object's methods is accomplished by sending messages to it. As one might expect, the object sending the message is called the **sender**, and the object receiving the message is called the **receiver**.

Armed with this basic knowledge about object oriented programming, let's look at some Ruby specifics.

Reading and Writing Ruby Code

Learning the syntax of a new language has the potential to induce the occasional yawn. So, to make things more interesting, I'll present it to you in a practical way that lets you play along at home: we'll use the **interactive Ruby shell**.

The Interactive Ruby Shell (`irb`)

You can fire up the interactive Ruby shell by entering `irb` into a terminal window.

Not the Standard DOS Box!

Windows users, don't forget to use the **Open Ruby Console Window** option from the Instant Rails control panel, to make sure the environment you're using contains the right settings.

`irb` allows you to issue Ruby commands interactively, one line at a time. This ability is great for playing with the language, and it's also handy for debugging, as we'll see in Chapter 11.

A couple of points about the `irb` output you'll see in this chapter:

- Lines beginning with the Ruby shell prompt (`irb>`) are typed in by the user.
- Lines beginning with `=>` show the return value of the command that has been entered.

We'll start with a *really* brief example:

```
irb> 1
=> 1
```

In this example, I've simply thrown the number `1` at the Ruby shell, and received back what appears to be the very same number.

Looks can be deceiving, though. It's actually *not* the very same number. What has been handed back is in fact a fully featured Ruby object.

Remember our discussion about object oriented programming in the previous section? Well, in Ruby, absolutely everything is treated as an object with which we can interact; each object belongs to a certain class, therefore each object is able to store data and functionality in the form of methods.

To find the class to which our number belongs, we call the number's `class` method:

```
irb> 1.class
=> Fixnum
```

We touched on senders and receivers earlier. In this example, we've sent the `class` message to the `1` object, so the `1` object is the receiver (there's no sender, as we're sending the message from the interactive command line rather than from another object). The value that's returned by the method we've invoked is `Fixnum`, which is the Ruby class that represents integer values.

Since everything in Ruby (*including* a class) is an object, we can actually send the very same message to the `Fixnum` class. The result is different, as we'd expect:

```
irb> Fixnum.class
=> Class
```

This time, the return value is `Class`, which is reassuring—we did invoke it on a classname, after all.

Note that the method `class` is all lowercase, yet the return value `Class` begins with a capital letter. A method in Ruby is always written in lowercase, whereas the first letter of a class is always capitalized.

Interacting with Ruby Objects

Becoming accustomed to thinking in terms of objects can take some time. Let's look at a few different types of objects, and see how we can interact with them.

Literal Objects

Literal objects are character strings or numbers that appear directly in the code, as did the number `1` that was returned in the previous section. We've seen numbers in action; next, let's look at a string literal.

A **string literal** is an object that contains a string of characters, such as a name, an address, or an especially witty phrase. In the same way that we created the `1` literal object in the previous example, we can easily create a new string literal object, then send it a message. A string literal is created by enclosing the characters that make up the string in single or double quotes, like this:

```
irb> "The quick brown fox"
=> "The quick brown fox"
```

First, we'll confirm that our string literal indeed belongs to class `String`:

```
irb> "The quick brown fox".class
=> String
```

This `String` object has a wealth of embedded functionality. For example, we can ascertain the number of characters that our string literal comprises by sending it the `length` message:

```
irb> "The quick brown fox".length
=> 19
```

Easy stuff, eh?

Variables and Constants

Every application needs a way to store information. Enter our variables and constants! As their names imply, these two data containers have their own unique roles to play.

A **constant** is an object that's assigned a value once, and once only (usually when the application starts up). Constants are therefore used to store information that doesn't need to change within a running application. As an example, a constant might be used to store the version number for an application. Constants in Ruby are always written using uppercase letters, as shown below:

```
irb> CONSTANT = "The quick brown fox in a constant"
=> "The quick brown fox in a constant"
irb> APP_VERSION = 5.04
=> 5.04
```

Variables, in contrast, are objects that are able to change at any time. They can even be reset to nothing, which frees up the memory space that they previously occupied. Variables in Ruby always start with a lowercase character:

```
irb> variable = "The quick brown fox in a variable"
=> "The quick brown fox in a variable"
```

There's one more special (and, you might say, *evil*) side to a variable: its **scope**. The scope of a variable is the part of the program to which a variable is visible. If you try to access a variable from outside its scope (for example, from a part of an application to which that variable is not visible), your attempts will generally fail.

The notable exception to the rules defining a variable's scope are global variables. As the name implies, a **global variable** is accessible from any part of the program. While this might sound convenient at first, usage of global variables is discouraged—the fact that they can be written to and read from any part of the program introduces security concerns.

Let's return to the string literal example we just saw. Assigning a `String` to a variable allows us to invoke on that variable the same methods we invoked on the string literal earlier:

```
irb> fox = "The quick brown fox"
=> "The quick brown fox"
irb> fox.class
=> String
irb> fox.length
=> 19
```

Punctuation in Ruby

The use of punctuation in Ruby code differs greatly from other languages such as Perl and PHP, so it can seem confusing at first if you're used to programming in those languages. However, once you have a few basics under your belt, punctuation in Ruby begins to feel quite intuitive and can greatly enhance the readability of your code.

Dot Notation

One of the most common punctuation characters in Ruby is the period (.). As we've seen, Ruby uses the period to separate the receiver from the message that's being sent to it, in the form *Object.receiver*.

If you need to comment a line, either for documentation purposes or to temporarily take a line of code out of the program flow, use a hash mark (#). Comments may start at the beginning of a line, or they may appear further along, after some Ruby code:

```
irb> # This is a comment. It doesn't actually do anything.
irb> 1 # So is this, but this one comes after a statement.
=> 1
irb> fox = "The quick brown fox"    # Assign to a variable
=> "The quick brown fox"
irb> fox.class                      # Display a variable's class
=> String
irb> fox.length                     # Display a variable's length
=> 19
```

Chaining Statements Together

Ruby doesn't require us to use any character to separate commands, unless we want to chain multiple statements together on a single line. In this case, a semicolon (;) is used as the separator. However, if you put every statement on its own line (as we've been doing until now), the semicolon is completely optional.

If you chain multiple statements together in the interactive shell, only the output of the last command that was executed will be displayed to the screen:

```
irb> fox.class; fox.length; fox.upcase
=> "THE QUICK BROWN FOX"
```

Use of Parentheses

If you ever delved into the source code of one of the many JavaScript libraries out there, you might have run screaming from your computer when you saw all the parentheses that are involved in the passing of arguments to methods.[3]

[3] http://www.sitepoint.com/article/javascript-library/

In Ruby, the use of parentheses for method calls is optional in cases in which no arguments are passed to the method. The following statements are therefore equal:

```
irb> fox.class()
=> String
irb> fox.class
=> String
```

It's common practice to include parentheses for method calls with multiple arguments, such as the `insert` method of the `String` class:

```
irb> "jumps over the lazy dog".insert(0, 'The quick brown fox ')
=> "The quick brown fox jumps over the lazy dog"
```

This call inserts the second argument passed to the `insert` object (`"The quick brown fox "`) at position `0` of the receiving `String` object (`"jumps over the lazy dog"`). Position `0` refers to the very beginning of the string.

Method Notation

Until now, we've looked at cases where Ruby uses *less* punctuation than its competitors. In fact, Ruby makes heavy use of expressive punctuation when it comes to the naming of methods.

A regular method name, as we've seen, is a simple, alphanumeric string of characters. If a method has a potentially destructive nature (for example, it directly modifies the receiving object rather than changing a copy of it), it's commonly suffixed with an exclamation mark (!).

The following example uses the `upcase` method to illustrate this point:

```
irb> fox.upcase
=> "THE QUICK BROWN FOX"
irb> fox
=> "The quick brown fox"
irb> fox.upcase!
=> "THE QUICK BROWN FOX"
irb> fox
=> "THE QUICK BROWN FOX"
```

Here, the contents of the `fox` variable have been modified by the `upcase!` method.

Punctuation is also used in the names of methods that return **boolean values**. A boolean value is a value that's either `true` or `false`; these values are commonly used as return values for methods that ask yes/no questions. Such methods end in a question mark, which nicely reflects the fact that they have yes/no answers:

```
irb> fox.empty?
=> false
irb> fox.is_a? String
=> true
```

These naming conventions make it easy to recognize methods that are destructive, and those that return boolean values, making your Ruby code more readable.

Object Oriented Programming in Ruby

Let's build on the theory that we covered at the start of this chapter as we take a look at Ruby's implementation of OOP.

As we already know, the structure of an application based on OOP principles is focused on interaction with objects. These objects are often representations of real-world objects, like a `Car`. Interaction with an object occurs when we send it a message or ask it a question. If we really did have a `Car` object called `kitt` (no, we don't—yet), starting the car might be as simple as doing this:

```
irb> kitt.start
```

This short line of Ruby code sends the message `start` to the object `kitt`. Using OOP terminology, we would say that this code statement calls the `start` method of the `kitt` object.

As I mentioned before, in contrast to other object oriented programming languages such as Python and PHP, in Ruby, *everything* is an object. Especially when compared with PHP, Ruby's OOP doesn't feel like a tacked-on afterthought—it was clearly intended to be a core feature of the language from the beginning, which makes using the OOP features in Ruby a real pleasure.

As we saw in the previous section, even the simplest of elements in Ruby (like literal strings and numbers) are objects to which you can send messages.

Classes and Objects

As in any other OOP language, in Ruby, each object belongs to a certain class (for example, pontiac_firebird might be an object of class Car). As we saw in the discussion at the beginning of this chapter, a class can group objects of a certain kind, and equip those objects with common functionality. This functionality comes in the form of methods, and in the object's ability to store information. For example, a pontiac_firebird object might need to store its mileage, as might any other object of the class Car.

In Ruby, the instantiation of a new object that's based on an existing class is accomplished by sending that class the new message. The result is a new object of that class. The following few lines of code show an extremely basic class definition in Ruby; the third line is where we create an instance of the class that we just defined:

```
irb> class Car
irb> end
=> nil
irb> kitt = Car.new
=> #<Car:0x75e54>
```

Another basic principle in OOP is **encapsulation**. According to this principle, objects should be treated as independent entities, each taking care of its own internal data and functionality. If we need to access an object's information—its internal variables, for instance—we make use of the object's **interface**, which is the subset of the object's methods that are made available for other objects to call.

Ruby provides objects with functionality at two levels—the object level, and class level—and adheres to the principle of encapsulation while it's at it! Let's dig deeper.

Object-level Functionality

At the object level, data storage is handled by **instance variables** (a name that's derived from the instantiation process mentioned above). Think of instance variables as storage containers that are attached to the object, but to which other objects do not have direct access.

To store or retrieve data from these variables, another object must call an **accessor method** on the object. An accessor method has the ability to set (and get) the value of the object's instance variables.

Let's look at how instance variables and accessor methods relate to each other, and how they're implemented in Ruby.

Instance Variables

Instance variables are bound to an object, and contain values for that object only.

Revisiting our car example, the mileage values for a number of different `Car` objects are likely to differ, as each car will have a different mileage. Therefore, `mileage` is held in an instance variable.

An instance variable can be recognized by its prefix: a single "at" sign (`@`). And what's more, instance variables don't even need to be declared! There's only one problem: we don't have any way to retrieve or change them from outside the object once they do exist. This is where instance methods come into play.

Instance Methods

Data storage and retrieval is not the only capability that can be bound to a specific object—functionality, too, can be bound to objects. We achieve this binding through the use of **instance methods**, which are specific to an object. Invoking an instance method (in other words, sending a message that contains the method name to an object) will invoke that functionality on the receiving object only.

Instance methods are defined using the `def` keyword, and end with the `end` keyword. Enter the following example into a new Ruby shell:

```
$ irb
irb> class Car
irb>   def open_trunk
irb>     # code to open trunk goes here
irb>   end
irb> end
=> nil
irb> kitt = Car.new
=> #<Car:0x75e54>
```

What you've done is define a class called `Car`, which has an instance method with the name `open_trunk`. A `Car` object instantiated from this class will—possibly using some fancy robotics connected to our Ruby program—open its trunk when its `open_trunk` method is called. Ignore that `nil` return value for the moment; we'll look at `nil` values in the next section.

Indenting Your Code

While the indentation of code is a key element of the syntax of languages such as Python, in Ruby, indentation is purely cosmetic—it aids readability, but does not affect the code in any way. In fact, while we're experimenting with the Ruby shell, you needn't be too worried about indenting any of the code. However, when we're saving files that will be edited later, you'll want the readability benefits that come from indenting nested lines.

The Ruby community has agreed upon two spaces as being optimum for indenting blocks of code such as class or method definitions. We'll adhere to this indentation scheme throughout this book.

With our class in place, we can make use of this method:

```
irb> kitt.open_trunk
=> nil
```

Since we don't want the trunks of all our cars to open at once, we've made this functionality available as an instance method.

I know, I know: we *still* haven't modified any data. We use accessor methods for this task.

Accessor Methods

An accessor method is a special type of instance method, and is used to read or write to an instance variable. There are two types: **readers** (sometimes called "getters") and **writers** (or "setters").

A reader method will look inside the object, fetch the value of an instance variable, and hand this value back to us. A writer method, on the other hand, will look inside the object, find an instance variable, and assign the variable the value that it was passed.

Let's add some methods for getting and setting the `@mileage` attribute of our `Car` objects. Once again, exit from the Ruby shell so that we can create an entirely new `Car` class definition. Our class definition is getting a bit longer now, so enter each line carefully. If you make a typing mistake, exit the shell and start over.

```
$ irb
irb> class Car
irb>   def set_mileage(x)
irb>     @mileage = x
irb>   end
irb>   def get_mileage
irb>     @mileage
irb>   end
irb> end
=> nil
irb> kitt = Car.new
=> #<Car:0x75e54>
```

Now, we can finally modify and retrieve the mileage of our `Car` objects!

```
irb> kitt.set_mileage(5667)
=> 5667
irb> kitt.get_mileage
=> 5667
```

This is still a bit awkward. Wouldn't it be nice if we could give our accessor methods exactly the same names as the attributes that they read from or write to? Luckily, Ruby contains shorthand notation for this very task. We can rewrite our class definition as follows:

```
$ irb
irb> class Car
irb>   def mileage=(x)
irb>     @mileage = x
irb>   end
irb>   def mileage
irb>     @mileage
irb>   end
irb> end
=> nil
irb> kitt = Car.new
=> #<Car:0x75e54>
```

With these accessor methods in place, we can read to and write from our instance variable as if it were available from outside the object:

```
irb> kitt.mileage = 6032
=> 6032
irb> kitt.mileage
=> 6032
```

These accessor methods form part of the object's interface.

Class-level Functionality

At the class level, **class variables** handle data storage. They're commonly used to store state information, or as a means of configuring default values for new objects. Class variables are typically set in the body of a class, and can be recognized by their prefix: a double "at" sign (`@@`).

First, enter the following class definition into a new Ruby shell:

```
$ irb
irb> class Car
irb>   @@number_of_cars = 0
irb>   def initialize
irb>     @@number_of_cars = @@number_of_cars + 1
irb>   end
irb> end
=> nil
```

The class definition for the class `Car` above has an internal counter for the total number of `Car` objects that have been created. Using the special instance method `initialize`, which is invoked automatically every time an object is instantiated, this counter is incremented for each new `Car` object.

By the way, we have actually already used a class method; I snuck it in there. The `new` method is an example of a class method that ships with Ruby and is available to all classes, whether they're defined by you or form part of the Ruby Standard Library.[4]

[4] The Ruby Standard Library is a large collection of classes that's included with every Ruby installation. The classes facilitate a wide range of common functionality, such as accessing web sites, date calculations, file operations, and more.

Custom class methods are commonly used to create objects with special properties (such as a default color for our Car objects), or to gather statistics about the class's usage.

Extending the earlier example, we could use a class method called count to return the value of the @@number_of_cars class variable. Remember that this is a variable that's incremented for every new Car object that's created. Class methods are defined identically to instance methods: using the def and end keywords. The only difference is that class method names are prefixed with self. Enter this code into a new Ruby shell:

```
$ irb
irb> class Car
irb>   @@number_of_cars = 0
irb>   def self.count
irb>     @@number_of_cars
irb>   end
irb>   def initialize
irb>     @@number_of_cars += 1
irb>   end
irb> end
=> nil
```

The following code instantiates some new Car objects, then makes use of our new class method:

```
irb> kitt = Car.new          # Michael Knight's talking car
=> #<0xba8c>
irb> herbie = Car.new        # The famous VolksWagen love bug!
=> #<0x8cd20>
irb> batmobile = Car.new     # Batman's sleek automobile
=> #<0x872e4>
irb> Car.count
=> 3
```

The method tells us that three instances of the Car class have been created. Note that we can't call a class method on an object.[5]

[5] Ruby actually does provide a way to invoke *some* class methods on an object, using the :: operator, but we won't worry about that for now. We'll see the :: operator in use in Chapter 4.

```
irb> kitt.count
NoMethodError: undefined method 'count' for #<Car:0x89da0>
```

As implied by the name, the `count` class method is available only to the `Car` class, not to any objects instantiated from that class.

I sneakily introduced something else in there. In many languages, including PHP and Java, the ++ and -- operators are used to increment a variable by one. Ruby doesn't support this notation; instead, when working with Ruby, we need to use the += operator. Therefore, the shorthand notation for incrementing our counter in the class definition is:

```
irb>     @@number_of_cars += 1
```

This code is identical to the following:

```
irb>     @@number_of_cars = @@number of cars + 1
```

Both of these lines can be read as "`my_variable` becomes equal to `my_variable` plus one."

Inheritance

If your application deals with more than the flat hierarchy we've explored so far, you may want to construct a scenario whereby some classes inherit from other classes. Continuing with the car analogy, let's suppose that we have a green limousine named Larry (this assignment of names to cars may seem a little strange, but it's important for this example, so bear with me). In Ruby, the `larry` object would probably descend from a `StretchLimo` class, which could in turn descend from the class `Car`. Let's implement that class relationship, to see how it works:

```
$ irb
irb> class Car
irb>   WHEELS = 4
irb> end
=> nil
irb> class StretchLimo < Car
irb>   WHEELS = 6
irb>   def turn_on_television
irb>     # Invoke code for switching on on-board TV here
irb>   end
irb> end
=> nil
```

Now, if we were to instantiate an object of class `StretchLimo`, we'd end up with a different kind of car. Instead of the regular four wheels that standard `Car` objects have, this one would have six wheels (stored in the class constant `WHEELS`). It would also have extra functionality, made possible by the presence of the extra method—`turn_on_television`—which would be available to be called by other objects.

However, if we were to instantiate a regular `Car` object, the car would have only four wheels, and there would be no instance method for turning on an on-board television. Think of inheritance as a way for the functionality of a class to become more specialized the further we move down the inheritance path.

Don't worry if you're struggling to wrap your head around all the aspects of OOP. You'll automatically become accustomed to them as you work through this book. You may find it useful to come back to this section, though, especially if you need a reminder about a certain term later on.

Return Values

It's always great to receive feedback. Remember our talk about passing arguments to methods? Well, regardless of whether or not a method accepts arguments, invoking a method in Ruby *always* results in feedback—it comes in the form of a return value, which is returned either explicitly or implicitly.

To return a value explicitly, use the `return` statement in the body of a method:

```
irb> def toot_horn
irb>   return "toooot!"
irb> end
=> nil
```

Calling the `toot_horn` method in this case would produce the following:

```
irb> toot_horn
=> "toooot!"
```

However, if no return statement is used, the result of the last statement that was executed is used as the return value. This behavior is quite unique to Ruby:

```
irb> def toot_loud_horn
irb>   "toooot!".upcase
irb> end
=> nil
```

Calling the `toot_loud_horn` method in this case would produce:

```
irb> toot_loud_horn
=> "TOOOOT!"
```

Standard Output

When you need to show output to the users of your application, use the `print` and `puts` ("put string") statements. Both methods will display the arguments passed to them as `String`s; `puts` also inserts a carriage return at the end of its output. Therefore, in a Ruby program the following lines:

```
print "The quick "
print "brown fox"
```

… would produce this output:

```
The quick brown fox
```

However, using `puts` like so:

```
puts "jumps over"
puts "the lazy dog"
```

… would produce this output:

```
jumps over
the lazy dog
```

At this stage, you might be wondering why *all* of the trial-and-error code snippets that we've typed into the Ruby shell actually produced output, given that we haven't been making use of the `print` or `puts` methods. The reason is that `irb` automatically writes the return value of the last statement it executes to the screen before displaying the `irb` prompt. This means that using a `print` or `puts` from within the Ruby shell might in fact produce two lines of output: the output that you specify should be displayed, and the return value of the last command that was executed, as in the following example:

```
irb> puts "The quick brown fox"
"The quick brown fox"
=> nil
```

Here, `nil` is actually the return value of the `puts` statement. Looking back at previous examples, you will have encountered `nil` as the return value for class and method definitions, and you'll have received a hexadecimal address, such as `#<Car:0x89da0>`, as the return value for object definitions. This hexadecimal value showed the location in memory that the object we instantiated occupied, but luckily we won't need to bother with such geeky details any further.

Having met the `print` and `puts` statements, you should be aware that a Rails application actually has a completely different approach to displaying output, called templates. We'll look at templates in Chapter 4.

Ruby Core Classes

We've already talked briefly about the `String` and `Fixnum` classes in the previous sections, but Ruby has a lot more under its hood. Let's explore!

Arrays

We use Ruby's `Arrays` to store collections of objects. Each individual object that's stored in an `Array` has a unique numeric key, which we can use to reference it. As with many languages, the first element in an `Array` is stored at position `0` (zero).

To create a new `Array`, simply instantiate a new object of class `Array` (using the `Array.new` construct). You can also use a shortcut approach, which is to enclose the objects you want to place inside the `Array` in square brackets.

For example, an `Array` containing the mileage at which a car is due for its regular service might look something like this:

```
irb> service_mileage = [5000, 15000, 30000, 60000, 100000]
=> [5000, 15000, 30000, 60000, 100000]
```

To retrieve individual elements from an `Array`, we specify the numeric key in square brackets:

```
irb> service_mileage[0]
=> 5000
irb> service_mileage[2]
=> 30000
```

Ruby has another shortcut, which allows us to create an `Array` from a list of `String`s: the `%w( )` syntax. Using this shortcut saves us from having to type a lot of double-quote characters:

```
irb> available_colors = %w( red green blue black )
=> ["red", "green", "blue", "black"]
irb> available_colors[0]
=> "red"
irb> available_colors[3]
=> "black"
```

In addition to facilitating simple element retrieval, `Array`s come with an extensive set of class methods and instance methods that ease data management tasks tremendously.

- `empty?` returns `true` if the receiving `Array` doesn't contain any elements:

  ```
  irb> available_colors.empty?
  => false
  ```

- `size` returns the number of elements in an `Array`:

  ```
  irb> available_colors.size
  => 4
  ```

- `first` and `last` return an `Array`'s first and last elements, respectively:

  ```
  irb> available_colors.first
  => "red"
  irb> available_colors.last
  => "black"
  ```

- `delete` removes the named element from the `Array` and returns it:

  ```
  irb> available_colors.delete "red"
  => "red"
  irb> available_colors
  => ["green", "blue", "black"]
  ```

The complete list of class methods and instance methods provided by the `Array` class is available via the Ruby reference documentation, which you can access by entering the `ri` command into the terminal window (for your operating system, *not* the Ruby shell), followed by the class name you'd like to look up:

```
$ ri Array
```

Oh, and `ri` stands for **r**uby **i**nteractive, in case you're wondering. Don't confuse it with `irb`.

Hashes

A `Hash` is another kind of data storage container. `Hash`es are similar, conceptually, to dictionaries: they map one object (the `key`: a word, for example) to another object (the `value`: a word's definition, for example) in a one-to-one relationship.

New `Hash`es can be created either by instantiating a new object of class `Hash` (using the `Hash.new` construct) or by using the curly brace shortcut shown below. When we define a `Hash`, we must specify each entry using the `key => value` syntax.

For example, the following `Hash` maps car names to a color:

```
irb> car_colors = {
irb>   'kitt' => 'black',
irb>   'herbie' => 'white',
irb>   'batmobile' => 'black',
irb>   'larry' => 'green'
irb> }
=> {"kitt"=>"black", "herbie"=>"white", "batmobile"=>"black",
   "larry"=>"green"}
```

To query this newly built `Hash`, we pass the key of the entry we want to look up in square brackets, like so:

```
irb> car_colors['kitt']
=> "black"
```

All sorts of useful functionality is built into `Hash`es, including the following methods:

- `empty?` returns `true` if the receiving `Hash` doesn't contain any elements:

  ```
  irb> car_colors.empty?
  => false
  ```

- `size` returns the number of elements in a `Hash`:

  ```
  irb> car_colors.size
  => 4
  ```

- `keys` returns all keys of a `Hash` as an `Array`:

  ```
  irb> car_colors.keys
  => ["kitt", "herbie", "batmobile", "larry"]
  ```

- `values` returns all values of a `Hash` as an `Array`, although care should be taken with regards to the order of the elements (keys in a `Hash` are ordered for optimal storage and retrieval; this order does not necessarily reflect the order in which they were entered):

```
irb> car_colors.values
=> ["black", "white", "black", "green"]
```

There are lots more class methods and instance methods provided by the `Hash` class. For a complete list, consult the Ruby reference documentation.

Strings

The typical Ruby `String` object—yep, that very same object we've been using in the past few sections—holds and manipulates sequences of characters. Most of the time, new `String` objects are created using string literals that are enclosed in single or double quotes. The literal can then be stored in a variable for later use:

```
irb> a_phrase = "The quick brown fox"
=> "The quick brown fox"
irb> a_phrase.class
=> String
```

If the string literal includes the quote character used to enclose the string itself, it must be escaped with a backslash character (\):

```
irb> 'I\'m a quick brown fox'
=> "I'm a quick brown fox"
irb> "Arnie said, \"I'm back!\""
=> "Arnie said, \"I'm back!\""
```

An easier way to specify string literals that contain quotes is to use the `%Q` shortcut, like this:

```
irb> %Q(Arnie said, "I'm back!")
=> "Arnie said, \"I'm back!\""
```

`String` objects also support the substitution of Ruby code into a string literal via the Ruby expression `#{}`:

```
irb> "The current time is: #{Time.now}"
=> "The current time is: Wed Aug 02 21:15:19 CEST 2006"
```

The `String` class also has rich embedded functionality for modifying `String` objects. Here are some of the most useful methods:

- `gsub` substitutes a given pattern within a `String`:

  ```
  irb> "The quick brown fox".gsub('fox', 'dog')
  => "The quick brown dog"
  ```

- `include?` returns `true` if a `String` contains another specific `String`:

  ```
  irb> "The quick brown fox".include?('fox')
  => true
  ```

- `length` returns the length of a `String` in characters:

  ```
  irb> "The quick brown fox".length
  => 19
  ```

- `slice` returns a portion of a `String`:

  ```
  irb> "The quick brown fox".slice(0, 3)
  => "The"
  ```

The complete method reference is available using the `ri` command line tool:

```
$ ri String
```

Numerics

Since there are so many different types of numbers, Ruby has a separate class for each, the popular `Float`, `Fixnum`, and `Bignum` classes among them. In fact, they're all subclasses of `Numeric`, which provides the basic functionality.

Just like `String`s, numbers are usually created from literals:

```
irb> 123.class
=> Fixnum
irb> 12.5.class
=> Float
```

Each of the specific `Numeric` subclasses comes with features that are relevant to the type of number it's designed to deal with. However, the following functionality is shared between all `Numeric` subclasses:

- `integer?` returns `true` if the object is a whole integer:

  ```
  irb> 123.integer?
  => true
  irb> 12.5.integer?
  => false
  ```

- `round` rounds a number to the nearest integer:

  ```
  irb> 12.3.round
  => 12
  irb> 38.8.round
  => 39
  ```

- `zero?` returns `true` if the number is equal to zero:

  ```
  irb> 0.zero?
  => true
  irb> 8.zero?
  => false
  ```

Additionally, there are ways to convert numbers between the `Numeric` subclasses. `to_f` converts a value to a `Float`, and `to_i` converts a value to an `Integer`:

```
irb> 12.to_f
=> 12.0
irb> 11.3.to_i
=> 11
```

Symbols

In Ruby, a `Symbol` is a simple textual identifier. Like a `String`, a `Symbol` is created using literals; the difference is that a `Symbol` is prefixed with a colon:

```
irb> :fox
=> :fox
irb> :fox.class
=> Symbol
```

The main benefit of using a `Symbol` instead of a `String` is that a `Symbol` contains less functionality. This can be an advantage in certain situations. For example, the `car_colors` `Hash` that we looked at earlier could be rewritten as follows:

```
car_colors = {
  :kitt => 'black',
  :herbie => 'white',
  :batmobile => 'black',
  :larry => 'green'
}
```

Objects of class `String` can be converted to `Symbol`s, and vice versa:

```
irb> "fox".to_sym
=> :fox
irb> :fox.to_s
=> "fox"
```

We'll use `Symbol`s frequently as we deal with Rails functionality in successive chapters of this book.

`nil`

I promised earlier that I'd explain `nil` values—now's the time!

All programming languages have a value that they can use when they actually mean *nothing*. Some use `undef`; others use `NULL`. Ruby uses `nil`. A `nil` value, like everything in Ruby, is also an object. It therefore has its own class: `NilClass`.

Basically, if a method doesn't return anything, it is, in fact, returning the value `nil`. And if you assign `nil` to a variable, you effectively make it empty. `nil` shows up in a couple of additional places, but we'll cross those bridges when we come to them.

Running Ruby Files

For the simple Ruby basics that we've experimented with so far, the interactive Ruby shell (`irb`) has been our tool of choice. I'm sure you'll agree that experimenting in a shell-like environment, where we can see immediate results, is a great way to learn the language.

However, we're going to be talking about control structures next, and for tasks of such complexity, you'll want to work in a text editor. This environment will allow you to run a chunk of code many times without having to retype it.

In general, Ruby scripts are simple text files containing Ruby code and have a **.rb** extension. These files are passed to the Ruby interpreter, which executes your code, like this:

```
$ ruby myscript.rb
```

To work with the examples that follow, I'd recommend that you open a new text file in your favorite text editor (which might be one of those I recommended back in Chapter 2) and type the code out as you go—this really is the best way to learn. However, I acknowledge that some people aren't interested in typing everything out, and just want to cut to the chase. For access to all of these examples, these more impatient readers can download the code archive for this book.[6] You can execute this code in the Ruby interpreter straight away.

As demonstrated above, to run the files from the command line, you simply need to type **`ruby`**, followed by the filename.

Control Structures

Ruby has a rich set of features for controlling the flow of your application. **Conditionals** are keywords that are used to decide whether or not certain statements are executed based on the evaluation of one or more conditions; **loops** are constructs

[6] http://www.sitepoint.com/books/rails2/code.php

that execute statements more than once; **blocks** are a means of encapsulating functionality (for example, to be executed in a loop).

To demonstrate these control structures, let's utilize some of the `Car` classes that we defined earlier. Type out the following class definition and save the file (or load it from the code archive); we'll build on it in this section as we explore some control structures:

01-car-classes.rb

```
class Car
  WHEELS = 4                  # class constant
  @@number_of_cars = 0        # class variable
  def initialize
    @@number_of_cars = @@number_of_cars + 1
  end
  def self.count
    @@number_of_cars
  end
  def mileage=(x)             # instance variable writer
    @mileage = x
  end
  def mileage                 # instance variable reader
    @mileage
  end
end

class StretchLimo < Car
  WHEELS = 6                  # class constant
  @@televisions = 1           # class variable
  def turn_on_television
    # Invoke code for switching on on-board TV here
  end
end

class PontiacFirebird < Car
end

class VolksWagen < Car
end
```

Conditionals

There are two basic conditional constructs in Ruby: `if` and `unless`. Each of these constructs can be used to execute a group of statements on the basis of a given condition.

The `if` Construct

An `if` construct wraps statements that are to be executed only if a certain condition is met. The keyword `end` defines the end of the `if` construct. The statements that are contained between the condition and the `end` keyword are executed only if the condition is met.

02-if-construct.rb *(excerpt)*

```
if Car.count.zero?
  puts "No cars have been produced yet."
end
```

You can provide a second condition by adding an `else` block: when the condition is met, the first block is executed; otherwise, the `else` block is executed. This kind of control flow will probably be familiar to you. Here it is in action:

03-if-else-construct.rb *(excerpt)*

```
if Car.count.zero?
  puts "No cars have been produced yet."
else
  puts "New cars can still be produced."
end
```

The most complicated example involves an alternative condition. If the first condition is not met, a second condition is evaluated. If neither conditions are met, the `else` block is executed:

04-if-elsif-else.rb *(excerpt)*

```
if Car.count.zero?
  puts "No cars have been produced yet."
elsif Car.count >= 10
  puts "Production capacity has been reached."
```

```
else
  puts "New cars can still be produced."
end
```

If the `count` method returned `5`, the code above would produce the following output:

```
New cars can still be produced.
```

An alternative to the traditional `if` condition is the `if` **statement modifier**. A statement modifier does just that: it modifies the statement of which it is part. The `if` statement modifier works exactly like a regular `if` condition, but it sits at the *end* of the line that's affected, rather than before a block of code:

05-if-statement-modifier.rb *(excerpt)*

```
puts "No cars have been produced yet." if Car.count.zero?
```

This version of the `if` condition is often used when the code that's to be executed conditionally comprises just a single line. Having the ability to create conditions like this results in code that's a lot more like English than other programming languages with more rigid structures.

The `unless` Construct

The `unless` condition is a negative version of the `if` condition. It's useful for situations in which you want to execute a group of statements when a certain condition is *not* met.

Let's create a few instances to work with:[7]

06-unless-construct.rb *(excerpt)*

```
kitt = PontiacFirebird.new
kitt.mileage = 5667
```

[7] Aficionados of comics will notice that I've visualized the BatMobile as a Pontiac Firebird—in fact, the caped crusader's choice of transport has varied over the years, taking in many of the automobile industry's less common innovations, and including everything from a 1966 Lincoln Futura to an amphibious tank. But we'll stick with a Pontiac for this example.

```
herbie = VolksWagen.new
herbie.mileage = 33014

batmobile = PontiacFirebird.new
batmobile.mileage = 4623

larry = StretchLimo.new
larry.mileage = 20140
```

Now if we wanted to find out how many Knight Rider fans KITT could take for a joyride, we could check which class the `kitt` object was. As with the `if` expression, the `end` keyword defines the end of the statement:

06-unless-construct.rb *(excerpt)*

```
unless kitt.is_a?(StretchLimo)
  puts "This car is only licensed to seat two people."
end
```

Like the `if` condition, the `unless` condition may have an optional `else` block of statements, which is executed when the condition is met:

07-unless-else.rb *(excerpt)*

```
unless kitt.is_a?(StretchLimo)
  puts "This car only has room for two people."
else
  puts "This car is licensed to carry up to 10 passengers."
end
```

Since KITT is definitely *not* a stretch limousine, this code would return:

```
This car only has room for two people.
```

Unlike `if` conditions, `unless` conditions do *not* support a second condition. However, like the `if` condition, the `unless` condition is also available as a statement modifier. The following code shows an example of this. Here, the message will not display if KITT's mileage is less than 25,000:

08-unless-statement-modifier.rb *(excerpt)*

```
puts "Service due!" unless kitt.mileage < 25000
```

Loops

Ruby provides the `while` and `for` constructs for looping through code (that is, executing a group of statements a specified number of times, or until a certain condition is met). Also, a number of instance methods are available for looping over the elements of an `Array` or `Hash`; we'll cover these in the next section.

`while` and `until` Loops

A `while` loop executes the statements it encloses repeatedly, as long as the specified condition is met:

09-while-loop.rb *(excerpt)*

```
while Car.count < 10
  Car.new
  puts "A new car instance was created."
end
```

This simple `while` loop executes the `Car.new` statement repeatedly, as long as the total number of cars is below 10. It exits the loop when the number reaches 10.

Like the relationship between `if` and `unless`, the `while` loop also has a complement: the `until` construct. If we use `until`, the code within the loop is executed *until* the condition is met. We could rewrite the loop above using `until` like so:

10-until-loop.rb *(excerpt)*

```
until Car.count == 10
  Car.new
  puts "A new car instance was created."
end
```

The Difference Between = and ==

It's important to note the difference between the **assignment operator** (a single equal sign) and the **equation operator** (a double equal sign) when using them within a condition.

If you're comparing two values, use the equation operator:

```
if Car.count == 10
  ⋮
end
```

If you're assigning a value to a variable, use the assignment operator:

```
my_new_car = Car.new
```

If you confuse the two, you might modify a value that you were hoping only to inspect, with potentially disastrous consequences!

`for` Loops

`for` loops allow us to iterate over the elements of a collection, such as an `Array`, and execute a group of statements once for each element. Here's an example:

11-for-loop.rb *(excerpt)*

```
for car in [ kitt, herbie, batmobile, larry ]
  puts car.mileage
end
```

The code above would produce the following output:

```
5667
33014
4623
20140
```

This simple `for` loop iterates over an `Array` of `Car` objects and outputs the `mileage` for each car. For each iteration, the `car` variable is set to the current element of the `Array`. The first iteration has `car` set to the equivalent of `kitt`, the second iteration has it set to `herbie`, and so forth.

In practice, the traditional `while` and `for` loops covered here are little used. Instead, most people tend to use the instance methods provided by the `Array` and `Hash` classes, which we'll cover next.

Blocks

Blocks are probably the single most attractive feature of Ruby. However, they're also one of those things that take a while to drop into place for Ruby newcomers. Before we dig deeper into creating blocks, let's take a look at some of the core features of Ruby that use blocks.

We looked at some loop constructs in the previous section, and this was a useful way to explore the tools that are available to us. However, you'll probably never actually come across many of these constructs in your work with other Ruby scripts, simply because it's almost always much easier to use a block to perform the same task. A block, in conjunction with the `each` method that is provided by the `Array` and `Hash` classes, is a very powerful way to loop through your data.

Let me illustrate this point with an example. Consider the `for` loop we used a moment ago. We could rewrite that code to use the `each` method, which is an instance method of the `Array` class, like so:

12-simple-block.rb ***(excerpt)***

```
[ kitt, herbie, batmobile, larry ].each do |car_name|
  puts car_name.mileage
end
```

Let's analyze this: the block comprises the code between the `do` and `end` keywords. A block is able to receive parameters, which are placed between vertical bars (`|`) at the beginning of the block. Multiple parameters are separated by commas. Therefore, this code performs an identical operation to the `for` loop we saw before, but in a much more succinct manner.

Let's take another example. To loop through the elements of a `Hash`, we use the `each` method, and pass two parameters to the block: the key (`car_name`) and the value (`color`):

13-block-with-params.rb *(excerpt)*

```
car_colors = {
  'kitt' => 'black',
  'herbie' => 'white',
  'batmobile' => 'black',
  'larry' => 'green'
}
car_colors.each do |car_name, color|
  puts "#{car_name} is #{color}"
end
```

This code produces the following output:

```
kitt is black
herbie is white
batmobile is black
larry is green
```

The `Integer` class also sports a number of methods that use blocks. The `times` method of an `Integer` object, for example, executes a block exactly *n* times, where *n* is the value of the object:

14-block-integer.rb *(excerpt)*

```
10.times { Car.new }
puts "#{Car.count} cars have been produced."
```

The code above produces this output:

```
10 cars have been produced.
```

One final point to note here is the alternative block syntax of curly braces. Instead of the `do…end` keywords that we used in previous examples, curly braces are the preferred syntax for blocks that are very short, as in the previous example.

Here's another method of the `Integer` class; in the spirit of `times`, the `upto` method counts from the value of the object up to the argument passed to the method:

15-block-upto.rb

```ruby
5.upto(7) { |i| puts i }
```

This code produces the output shown here:

```
5
6
7
```

In Ruby parlance, the object `i` is a parameter of the block. Parameters for blocks are enclosed in vertical bars, and are usually available only from within the block. If we have more than one parameter, we separate them using commas, like so: `|parameter1, parameter2|`. In the example above, we would no longer have access to `i` once the block had finished executing.

As we work through this book, we'll explore many more uses of blocks in combination with the Rails core classes.

Summary

Wow, we covered a lot in this chapter! First, we swept through a stack of object oriented programming theory—probably the equivalent of an introductory computer science course! This gave us a good grounding for exploring the basics of the Ruby programming language, and the Interactive Ruby Shell (`irb`) was a fun way to do this exploration.

We also investigated many of the Ruby core classes, such as `String`, `Array`, and `Hash`, from within the Ruby shell. We then moved from the shell to create and save proper Ruby files, and using these files, we experimented with control structures such as conditionals, loops, and blocks.

In the next chapter, we'll look at the major cornerstones that make up the Rails framework—the integrated testing facilities—as well as the roles played by the development, testing, and production environments.

Chapter 4

Rails Revealed

As you'll have gathered from Chapter 1, quite a bit of thought has been put into the code base that makes up the Rails framework. Over time, many of the internals have been rewritten, which has improved their speed and efficiency and allowed the implementation of additional features, but the original architecture remains largely unchanged. This chapter will shed some light on the inner workings of Rails.

Three Environments

Rails encourages the use of a different environment for each of the stages in an application's life cycle—development, testing, and production. If you've been developing web applications for a while, this is probably how you operate anyway; Rails just formalizes these environments.

development

In the development environment, changes to an application's source code are immediately visible; all we need to do is reload the corresponding page in a web browser. Speed is not a critical factor in this environment. Instead, the focus is on providing the developer with as much insight as possible into the components responsible for displaying each page. When an error occurs in the development environment, we are able to tell at a glance which line of code is responsible for the error, and how that particular line was invoked. This capability is provided by the **stack trace** (a comprehensive list of all the method calls leading up to the error), which is displayed when an unexpected error occurs.

test

In testing, we usually refresh the database with a baseline of dummy data each time a test is repeated: this step ensures that the results of the tests are consistent, and that behavior is reproducible. Unit and functional testing procedures are fully automated in Rails.

When we test a Rails application, we don't view it using a traditional web browser. Instead, tests are invoked from the command line, and can be run as background processes. The testing environment provides a dedicated space in which these processes can operate.

production

By the time your application finally goes live, it should be sufficiently tested that all—or at least most—of the bugs have been eliminated. As a result, updates to the code base should be infrequent, which means that the production environments can be optimized to focus on performance. Tasks such as writing extensive logs for debugging purposes should be unnecessary at this stage. Besides, if an error does occur, you don't want to scare your visitors away with a cryptic stack trace; that's best kept for the development environment.

As the requirements of each of the three environments are quite different, Rails stores the data for each environment in entirely separate databases. So at any given time, you might have:

- live data with which real users are interacting in the production environment
- a partial copy of this live data that you're using to debug an error or develop new features in the development environment
- a set of testing data that's constantly being reloaded into the testing environment

Let's look at how we can configure our database for each of these environments.

Database Configuration

Configuring the database for a Rails application is incredibly easy. All of the critical information is contained in just one file. We'll take a close look at this database configuration file, then create some databases for our application to use.

The Database Configuration File

The separation of environments is reflected in the Rails database configuration file **database.yml**. We saw a sample of this file back in Chapter 1, and in fact we created our very own configuration file in Chapter 2, when we used the `rails` command. Go take a look—it lives in the **config** subdirectory of our Shovell application.

With the comments removed, the file should look like this:

01-database.yml

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  timeout: 5000

test:
  adapter: sqlite3
  database: db/test.sqlite3
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  timeout: 5000
```

This file lists the minimum amount of information we need in order to connect to the database server for each of our environments (development, test, and production). With the default setup of SQLite that we installed in Chapter 2, every environment is allocated its own physically separate database file, which calls the **db** subdirectory home.

The parameter `database` sets the name of the database that is to be used in each environment. As the configuration file suggests, Rails can support multiple databases (and even different types of database engines, such as MySQL for production and

SQLite for development) in parallel. Note that we're actually talking about different *databases* here, not just different tables—each database can host an arbitrary number of different tables in parallel.

Figure 4.1 shows a graphical representation of this architecture.

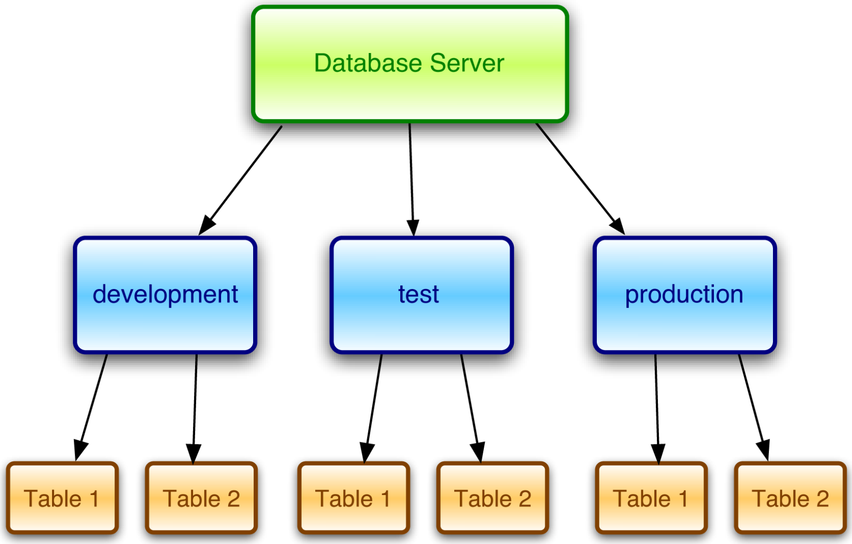


Figure 4.1. The database architecture of a Rails application

However, there's one startling aspect missing from our current configuration: looking at the **db** subdirectory, the databases referenced in our configuration file don't exist yet! Fear not, Rails will auto-create them as soon as they're needed. There's nothing we need to do as far as they are concerned.

The Model-View-Controller Architecture

The model-view-controller (MVC) architecture that we first encountered in Chapter 1 is not unique to Rails. In fact, it predates both Rails and the Ruby language by many years. However, Rails really takes the idea of separating an application's data, user interface, and control logic to a whole new level.

Let's take a look at the concepts behind building an application using the MVC architecture. Once we have the theory in place, we'll see how it translates to our Rails code.

MVC in Theory

MVC is a pattern for the architecture of a software application. It separates an application into the following three components:

models
: for handling data and business logic

controllers
: for handling the user interface and application logic

views
: for handling graphical user interface objects and presentation logic

This separation results in user requests being processed as follows:

1. The browser, on the client, sends a request for a page to the controller on the server.
2. The controller retrieves the data it needs from the model in order to respond to the request.
3. The controller hands the retrieved data to the view.
4. The view is rendered and sent back to the client for the browser to display.

This process is illustrated in Figure 4.2.

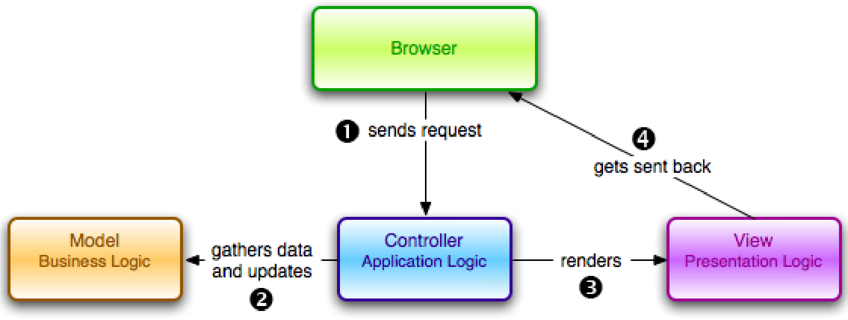


Figure 4.2. Processing a page request in an MVC architecture

Separating a software application into these three distinct components is a good idea for a number of reasons, including the following:

It improves scalability (the ability for an application to grow).
: For example, if your application begins experiencing performance issues because database access is slow, you can upgrade the hardware running the database without other components being affected.

It makes maintenance easier.
: As the components have a low dependency on each other, making changes to one (to fix bugs or change functionality) does not affect another.

It promotes reuse.
: A model may be reused by multiple views, and vice versa.

If you haven't quite got your head around the concept of MVC yet, don't worry. For now, the important thing is to remember that your Rails application is separated into three distinct components. Jump back to Figure 4.2 if you need to refer to it later on.

MVC the Rails Way

Rails promotes the concept that models, views, and controllers should be kept quite separate by storing the code for each of these elements as separate files in separate directories.

This is where the Rails directory structure that we created back in Chapter 2 comes into play. The time has come for us to poke around a bit within that structure. If you take a look inside the **app** directory, which is depicted in Figure 4.3, you'll see some folders whose names might be starting to sound familiar.

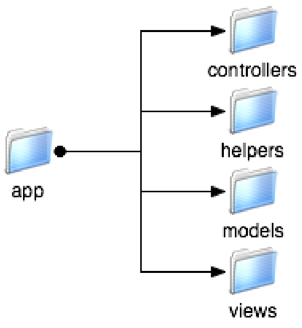


Figure 4.3. The app subdirectory

As you can see, each component of the model-view-controller architecture has its place within the **app** subdirectory—the **models**, **views**, and **controllers** subdirectories, respectively. (We'll talk about that **helpers** directory in Chapter 6.)

This separation continues within the code that comprises the framework itself. The classes that form the core functionality of Rails reside within the following modules:

`ActiveRecord`

`ActiveRecord` is the module for handling business logic and database communication. It plays the role of model in our MVC architecture.[1]

`ActionController`

`ActionController` is the component that handles browser requests and facilitates communication between the model and the view. Your controllers will inherit from this class. It forms part of the `ActionPack` library, a collection of Rails components that we'll explore in depth in Chapter 5.

`ActionView`

`ActionView` is the component that handles the presentation of pages returned to the client. Views inherit from this class, which is also part of the `ActionPack` library.

[1] While it might seem odd that `ActiveRecord` doesn't have the word "model" in its name, there is a reason for this: Active Record is also the name of a famous design pattern—one that this component implements in order to perform its role in the MVC world. Besides, if it had been called `ActionModel` then it would have sounded more like an overpaid Hollywood star than a software component …

Let's take a closer look at each of these components in turn.

`ActiveRecord` (the Model)

`ActiveRecord` is designed to handle all of an application's tasks that relate to the database, including:

- establishing a connection to the database server
- retrieving data from a table
- storing new data in the database

`ActiveRecord` also has a few other neat tricks up its sleeve. Let's look at some of them now.

Database Abstraction

`ActiveRecord` ships with database adapters to connect to SQLite, MySQL, and PostgreSQL. A large number of adapters are also available for other popular database server packages, such as Oracle, DB2, and Microsoft SQL Server, via the RubyGems system.

The `ActiveRecord` module is based on the concept of database abstraction. As we mentioned in Chapter 1, database abstraction is a way of coding an application so that it isn't dependent upon any one database. Code that's specific to a particular database server is hidden safely in `ActiveRecord`, and invoked as needed. The result is that a Rails application is not bound to any specific database server software. Should you need to change the underlying database server at a later time, no changes to your application code should be required.

Examples of code that differs greatly between vendors, and which `ActiveRecord` abstracts, include:

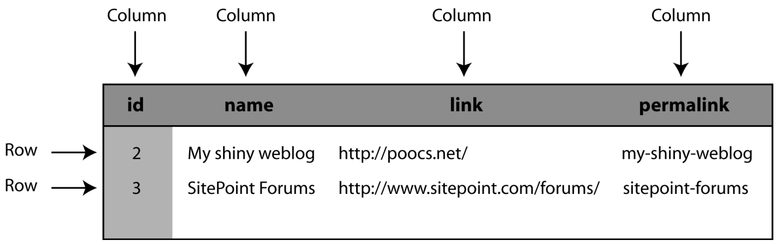
- the process of logging into the database server
- date calculations
- handling of boolean (`true`/`false`) data
- evolution of your database structure

Before I can show you the magic of `ActiveRecord` in action, though, we need to do a little housekeeping.

Database Tables

Tables are the containers within a database that store our data in a structured manner, and they're made up of rows and columns. The rows map to individual objects, and the columns map to the attributes of those objects. The collection of all the tables in a database, and the relationships between those tables, is called the database **schema**.

An example of a table is shown in Figure 4.4.



id	name	link	permalink
2	My shiny weblog	http://poocs.net/	my-shiny-weblog
3	SitePoint Forums	http://www.sitepoint.com/forums/	sitepoint-forums

Figure 4.4. The structure of a typical database table, including rows and columns

In Rails, the naming of Ruby classes and database tables follows an intuitive pattern: if we have a table called `stories` which consists of five rows, this table will store the data for five `Story` objects. The nicest thing about the mapping between classes and tables is that you don't need to write code to achieve it—the mapping just happens, because `ActiveRecord` infers the name of the table from the name of the class.

Note that the name of our class in Ruby is a singular noun (`Story`), but the name of the table is plural (`stories`). This relationship makes sense if you think about it: when we refer to a `Story` object in Ruby, we're dealing with a single story. But the SQL table holds a multitude of stories, so its name should be plural. While you can override these conventions—as is sometimes necessary when dealing with legacy databases—it's much easier to adhere to them.

The close relationship between tables and objects extends even further: if our `stories` table were to have a `link` column, as our example in Figure 4.4 does, the data in this column would automatically be mapped to the `link` attribute in a `Story` object. And adding a new column to a table would cause an attribute of the same name to become available in all of that table's corresponding objects.

So, let's create some tables to hold the stories we create.

For the time being, we'll create a table using the old-fashioned approach of entering SQL into the SQLite console. You could type out the following SQL commands, although typing out SQL isn't much fun. Instead, I encourage you to download the following script from the code archive, and copy and paste it straight into your SQLite console that you invoked via the following command in the application directory:

```
$ sqlite3 db/development.sqlite3
```

Once your SQLite console is up, paste in the following:

02-create-stories-table.sql

```
CREATE TABLE stories (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  "name" varchar(255) DEFAULT NULL,
  "link" varchar(255) DEFAULT NULL,
  "created_at" datetime DEFAULT NULL,
  "updated_at" datetime DEFAULT NULL
);
```

You needn't worry about remembering these SQL commands to use in your own projects; instead, take heart in knowing that in Chapter 5 we'll look at **migrations**. Migrations are special Ruby classes that we can write to create database tables for our application without using any SQL at all.

Using the Rails Console

Now that we have our `stories` table in place, let's exit the SQLite console (simply type `.quit`) and open up a Rails console. A Rails console is just like the interactive Ruby console (`irb`) that we used in Chapter 3, but with one key difference. In a Rails console, you have access to all the environment variables and classes that are available to your application while it's running. These are not available from within a standard `irb` console.

To enter a Rails console, change to your **shovell** folder, and enter the command `ruby script/console`, as shown below. The >> prompt is ready to accept your commands:

```
$ cd shovell
$ ruby script/console
Loading development environment (Rails 2.0.2)
>>
```

Saving an Object

To start using `ActiveRecord`, simply define a class that inherits from the `ActiveRecord::Base` class. (We touched on the `::` operator very briefly in Chapter 3, where we mentioned that it was a way to invoke class methods on an object. It can also be used to refer to classes that exist within a module, which is what we're doing here.) Flip back to the section on object oriented programming (OOP) in Chapter 3 if you need a refresher on inheritance.

Consider the following code snippet:

```
class Story < ActiveRecord::Base
end
```

These two lines of code define a seemingly empty class called `Story`. However, this class is far from empty, as we'll soon see.

From the Rails console, let's create this `Story` class, and an instance of the class called `story`, by entering these commands:

```
>> class Story < ActiveRecord::Base; end
=> nil
>> story = Story.new
=> #<Story id: nil, name: nil, url: nil, created_at: nil,
      updated_at: nil>
>> story.class
=> Story(id: integer, name: string, link: string,
      created_at: datetime, updated_at: datetime)
```

As you can see, the syntax for creating a new `ActiveRecord` object is identical to the syntax we used to create other Ruby objects in Chapter 3. At this point, we've created a new `Story` object. However, this object exists in memory only—we haven't stored it in our database yet.

We can confirm the fact that our `Story` object hasn't been saved yet by checking the return value of the `new_record?` method:

```
>> story.new_record?
=> true
```

Since the object has not been saved yet, it will be lost when we exit the Rails console. To save it to the database, we need to invoke the object's `save` method:

```
>> story.save
=> true
```

Now that we've saved our object (a return value of `true` indicates that the `save` method was successful) our story is no longer a new record. It's even been assigned a unique ID, as shown below:

```
>> story.new_record?
=> false
>> story.id
=> 1
```

Defining Relationships Between Objects

As well as the basic functionality that we've just seen, `ActiveRecord` makes the process of defining relationships (or associations) between objects as easy as it can be. Of course, it's possible with some database servers to define such relationships entirely within the database schema. However, in order to put `ActiveRecord` through its paces, let's look at the way it defines these relationships within Rails.

Object relationships can be defined in a variety of ways; the main difference between these relationships is the number of records that are specified in the relationship. The primary types of database associations are:

- one-to-one associations
- one-to-many associations
- many-to-many associations

Let's look at some examples of each of these associations. Feel free to type them into the Rails console if you like, for the sake of practice. Remember that your class definitions won't be saved, though—I'll show you how to define associations in a file later.

Suppose our application has the following associations:

- An Author can have one Weblog:

  ```
  class Author < ActiveRecord::Base
    has_one :weblog
  end
  ```

- An Author can submit many Stories:

  ```
  class Author < ActiveRecord::Base
    has_many :stories
  end
  ```

- A Story belongs to an Author:

  ```
  class Story < ActiveRecord::Base
    belongs_to :author
  end
  ```

- A Story has, and belongs to, many different Topics:

  ```
  class Story < ActiveRecord::Base
    has_and_belongs_to_many :topics
  end
  class Topic < ActiveRecord::Base
    has_and_belongs_to_many :stories
  end
  ```

You're no doubt growing tired of typing class definitions into a console, only to have them disappear the moment you exit the console. For this reason, we won't go any further with the associations between our objects—we'll delve into the Rails ActiveRecord module in more detail in Chapter 5.

The ActionPack Module

ActionPack is the name of the library that contains the view and controller parts of the MVC architecture. Unlike the ActiveRecord module, these modules are a little more intuitively named: ActionController and ActionView.

Exploring application logic and presentation logic on the command line doesn't make a whole lot of sense; views and controllers *are* designed to interact with a web browser, after all! Instead, I'll just give you a brief overview of the `ActionPack` components, and we'll cover the hands-on stuff in Chapter 5.

`ActionController` (the Controller)

The controller handles the application logic of your program, acting as glue between the application's data, the presentation layer, and the web browser. In this role, a controller performs a number of tasks, including:

- deciding how to handle a particular request (for example, whether to render a full page or just one part of it)
- retrieving data from the model to be passed to the view
- gathering information from a browser request, and using it to create or update data in the model

When we introduced the MVC diagram in Figure 4.2 earlier in this chapter, it might not have occurred to you that a Rails application can consist of a number of different controllers. Well, it can! Each controller is responsible for a specific part of the application.

For our Shovell application, we'll create:

- one controller for displaying story links, which we'll name `StoriesController`
- another controller for handling user authentication, called `SessionsController`
- a controller to display user pages, named `UsersController`
- and finally a fourth controller to handle story voting, called `VotesController`

All controllers will inherit from the `ActionController::Base` class,[2] but they'll have different functionality, implemented as instance methods. Here's a sample class definition for the `StoriesController` class:

[2] There will actually be an intermediate class between this class and the `ActionController::Base` class; we'll cover the creation of the `StoriesController` class in more detail in Chapter 5. However, this doesn't change the fact that `ActionController::Base` is the base class from which every controller inherits.

```
class StoriesController < ActionController::Base
  def index
  end

  def show
  end
end
```

This simple class definition sets up our `StoriesController` with two empty methods: the `index` method, and the `show` method. We'll expand upon both of these methods in later chapters.

Each controller resides in its own Ruby file (with a **.rb** extension), which lives within the **app/controllers** directory. The `StoriesController` class that we just defined, for example, would inhabit the file **app/controllers/stories_controller.rb**.

Naming Classes and Files

You'll have noticed by now that the names of classes and files follow different conventions:

- Class names are written in **CamelCase** (each word beginning with a capital letter, with no spaces between words).[3]
- Filenames are written in lowercase, with underscores separating each word.

This is an important detail! If this convention is *not* followed, Rails will have a hard time locating your files. Luckily, you won't need to name your files manually very often, if ever, as you'll see when we look at generated code in Chapter 5.

`ActionView` (the View)

As we discussed earlier, one of the principles of MVC is that a view should contain **presentation logic** only. This principle holds that the code in a view should only perform actions that relate to displaying pages in the application—none of the code in a view should perform any complicated application logic, nor should it store or

[3] There are actually two variations of CamelCase: one with an uppercase first letter (also known as PascalCase), and one with a lowercase first letter. The Ruby convention for class names requires an uppercase first letter.

retrieve any data from the database. In Rails, everything that is sent to the web browser is handled by a view.

Predictably, views are stored in the **app/views** folder of our application.

A view need not actually contain any Ruby code at all—it may be the case that one of your views is a simple HTML file. However, it's more likely that your views will contain a combination of HTML and Ruby code, making the page more dynamic. The Ruby code is embedded in HTML using **embedded Ruby** (ERb) syntax.

ERb is similar to PHP or JSP in that it allows server-side code to be scattered throughout an HTML file by wrapping that code in special tags. For example, in PHP you may write code like this:

```
<strong><?php echo 'Hello World from PHP!' ?></strong>
```

The equivalent code in ERb would be the following:

```
<strong><%= 'Hello World from Ruby!' %></strong>
```

There are two forms of the ERb tag pair: one that includes the equal sign, and one that doesn't:

`<%= … %>`
: This tag pair is for regular output. The output of a Ruby expression between these tags will be displayed in the browser.

`<% … %>`
: This tag pair is for code that is not intended to be displayed, such as calculations, loops, or variable assignments.

An example of each ERb tag is shown below:

```
<%= 'This line is displayed in the browser' %>
<% 'This line executes silently, without displaying any output' %>
```

You can place any Ruby code—be it simple or complex—between these tags.

Creating an instance of a view is a little different to that of a model or a controller. While `ActionView::Base` (the parent class for all views) is one of the base classes for views in Rails, the instantiation of a view is handled completely by the `ActionView` module. The only file a Rails developer needs to modify is the **template**, which is the file that contains the presentation code for the view. As you might have guessed, these templates are stored in the **app/views** folder.

As with everything else Rails, a strict convention applies to the naming and storage of template files:

- A template has a one-to-one mapping to the action (method) of a controller. The name of the template file matches the name of the action to which it maps.
- The folder that stores the template is named after the controller.
- The extension of the template file is twofold and varies depending on the template's type and the actual language in which a template is written. By default there are three types of extensions in Rails:

  html.erb
  : This is the extension for standard HTML templates that are sprinkled with ERb tags.

  xml.builder
  : This extension is used for templates that output XML (for example, to generate RSS feeds for your application).

  js.rjs
  : This extension is used for templates that return JavaScript instructions. This type of template might be used, for example, to modify an existing page (via Ajax) to update the contents of a `<div>` tag.

This convention may sound complicated, but it's actually quite intuitive. For example, consider the `StoriesController` class defined earlier. Invoking the `show` method for this controller would, by default, attempt to display the `ActionView` template that lived in the **app/views/stories** directory. Assuming the page was a standard HTML page (containing some ERb code), the name of this template would be **show.html.erb**.

Rails also comes with special templates such as **layouts** and **partials**. Layouts are templates that control the global layout of an application, such as structures that remain unchanged between pages (the primary navigation menu, for instance).

Partials are special subtemplates (the result of a template being split into separate files, such as a secondary navigation menu or a form) that can be used multiple times within the application. We'll cover both layouts and partials in Chapter 7.

Communication between controllers and views occurs via instance variables that are populated from within the controller's action. Let's expand upon our sample `StoriesController` class to illustrate this point (there's no need to type any of this out just yet):

```
class StoriesController < ActionController::Base
  def index
    @variable = 'Value being passed to a view'
  end
end
```

As you can see, the instance variable `@variable` is being assigned a string value within the controller's action. Through the magic of `ActionView`, this variable can now be referenced directly from the corresponding view, as shown in the code below:

```
<p>The instance variable @variable contains: <%= @variable %></p>
```

This approach allows more complex computations to be performed outside the view—remember, it should only contain presentational logic—and allow the view to display just the end result of the computation.

Rails also provides access to special containers, such as the `params` and `session` hashes. These contain such information as the current page request and the user's session. We'll make use of these hashes in the chapters that follow.

The REST

When I introduced Rails in Chapter 1 I mentioned quite a few common development principles and best practices that the Rails team advises you to adopt in your own projects. One that I kept under my hat until now was RESTful-style development, or resource-centric development. REST will make much more sense with your fresh knowledge about models and controllers as the principal building blocks of a Rails application.

In Theory

REST stands for *Re*presentational *S*tate *T*ransfer and originates from the doctoral dissertation of Roy Fielding,[4] one of the co-founders of The Apache Software Foundation and one of the authors of the HTTP specification.

REST, according to the theory, is not restricted to the World Wide Web. The basis of the resource-centric approach is derived from the fact that most of the time spent using network-based applications can be characterized as a client or user interacting with distinct resources. For example, in an ecommerce application, a book and a shopping cart are separate resources with which the customer interacts.

Every resource in an application needs to be addressed by a unique and uniform identifier. In the world of web applications, the unique identifier would be the URL by which a resource can be accessed. In our Shovell application, each submitted story will be able to be viewed at a unique URL.

The potential interactions within an application are defined as a set of operations (or verbs) that can be performed with a given resource. The most common verbs are *c*reate, *r*ead, *u*pdate, and *d*elete, which are often collectively referred to as "*CRUD* operations." If you relate this to our Shovell application you'll see that it covers most of the interactions possible with the Shovell stories: a user will create a story, another user will read the story, the story can also be updated or deleted.

The client and server have to communicate via the same language (or protocol) in order to implement the REST architecture style successfully. This protocol is also required to be **stateless**, **cacheable**, and **layered**.

Here, stateless means that each request for information from the client to the server needs to be completely independent of prior or future requests. Each request needs to contain everything necessary for the server to understand the request and provide an appropriate answer.

Cacheable and layered are architectural attributes that improve the communication between client and server without affecting the communication protocol.

[4] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

REST and the Web

As mentioned in the previous section, REST as a design pattern can be used in any application domain. But the Web is probably the domain that implements REST most often. Since this is a book that deals with building web applications, we'd better take a look at the implementation details of RESTful style development in web applications in particular.

HTTP (Hypertext Transfer Protocol: the communication protocol used on the Web), as the astute reader will know, also makes heavy use of verbs in its day-to-day operations. When your browser requests a web page from any given web server, it will issue a so-called `GET`-request. If you submit a web page form, your browser will do so using a `POST`-request (not always, to be honest, but 99% of the time).

In addition to `GET` and `POST`, HTTP defines two additional verbs that are less commonly used by web browsers. (In fact, none of the browsers in widespread use actually implement them.) These verbs are `PUT` and `DELETE`. If you compare the list of HTTP verbs with the verbs of CRUD from the previous section, they line up fairly nicely, as you can see in Table 4.1.

Table 4.1. HTTP Verbs Versus CRUD Verbs

CRUD	HTTP
`CREATE`	`POST`
`READ`	`GET`
`UPDATE`	`PUT`
`DELETE`	`DELETE`

The language in which client (the browser) and server (the web server) talk to each other is obviously HTTP. HTTP is, by definition, stateless. This means that as soon as a browser has downloaded all of the information the server offered as a reply to the browser's request, the connection is closed and the two might never ever talk again. Or the browser could send another request just milliseconds later, asking for additional information. Each request contains all the necessary information for the server to respond appropriately, including potential cookies, the format, and the language in which the browser expects the server to reply.

HTTP is also layered and cacheable, both of which are attributes the REST definition expects of the spoken protocol. Routers, proxy servers, and firewalls are only three (very common) examples of architectural components that implement layering and caching on top of HTTP.

REST in Rails

REST and Rails not only both start with the letter R, they have a fairly deep relationship. Rails comes with a generator for resources (see the section called "Code Generation" for a primer on this topic) and provides all sorts of assistance for easy construction of the uniform addresses by which resources can be accessed.

Rails's focus on the MVC architecture (which we'll be getting our hands on shortly, in Chapter 5) is also a perfect companion for RESTful style development. Models resemble the resources themselves, while controllers provide access to the resource and allow interaction based on the interaction verbs listed above.

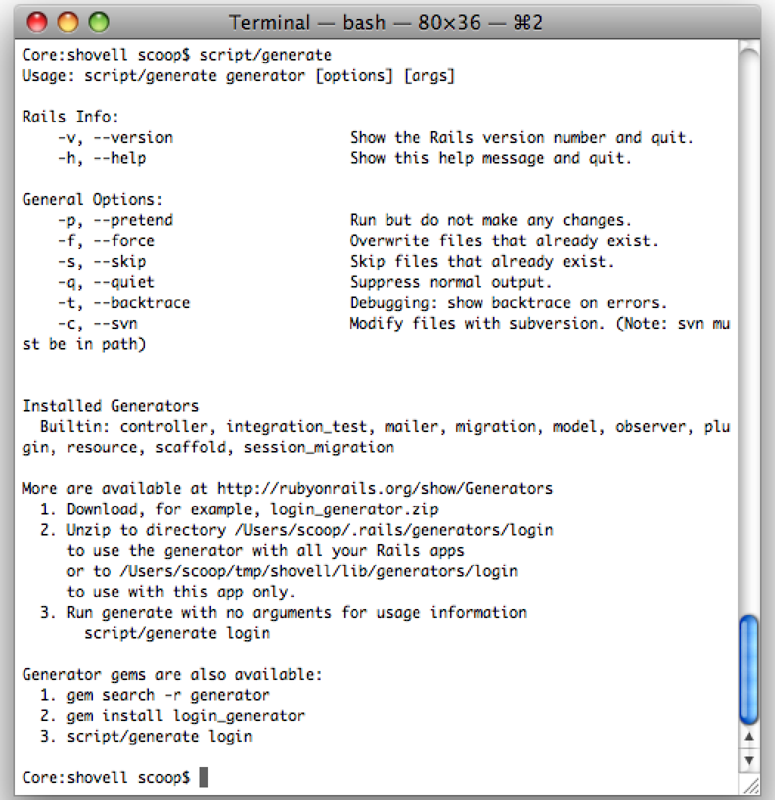
I mentioned in the previous section that two verbs aren't implemented in the majority of browsers on the market. To support the verbs `PUT` and `DELETE`, Rails uses `POST` requests with a little tacked-on magic to simulate the `PUT` and `DELETE` verbs transparently for both the user and the Rails application developer. Nifty, isn't it?

We will gradually start implementing and interacting with resources for our Shovell application over the course of the next hands-on chapters, so let's now move on and talk about yet another batch of components that make up the Rails framework.

Code Generation

Rather than having us create all of our application code from scratch, Rails gives us the facility to generate an application's basic structure with considerable ease. In the same way that we created our application's entire directory structure, we can create new models, controllers, and views using a single command.

To generate code in Rails, we use the **generate** script, which lives in the **script** folder. Give it a try now: type `ruby script/generate` without any command parameters. Rails displays an overview of the available parameters for the command, and lists the generators from which we can choose, as Figure 4.5 illustrates.



```
Core:shovell scoop$ script/generate
Usage: script/generate generator [options] [args]

Rails Info:
    -v, --version                    Show the Rails version number and quit.
    -h, --help                       Show this help message and quit.

General Options:
    -p, --pretend                    Run but do not make any changes.
    -f, --force                      Overwrite files that already exist.
    -s, --skip                       Skip files that already exist.
    -q, --quiet                      Suppress normal output.
    -t, --backtrace                  Debugging: show backtrace on errors.
    -c, --svn                        Modify files with subversion. (Note: svn mu
st be in path)


Installed Generators
  Builtin: controller, integration_test, mailer, migration, model, observer, plu
gin, resource, scaffold, session_migration

More are available at http://rubyonrails.org/show/Generators
  1. Download, for example, login_generator.zip
  2. Unzip to directory /Users/scoop/.rails/generators/login
     to use the generator with all your Rails apps
     or to /Users/scoop/tmp/shovell/lib/generators/login
     to use with this app only.
  3. Run generate with no arguments for usage information
       script/generate login

Generator gems are also available:
  1. gem search -r generator
  2. gem install login_generator
  3. script/generate login

Core:shovell scoop$ 
```

Figure 4.5. Sample output from `script/generate`

Rails can generate code of varying complexity. At its simplest, creating a new controller causes a template file to be placed in the appropriate subdirectory of your application. The template itself consists of a mainly empty class definition, similar to the `Story` and `Author` classes that we looked at earlier in this chapter.

However, code generation can also be a very powerful tool for automating complex, repetitive tasks; for instance, you might generate a foundation for handling user authentication. We'll launch straight into generating code in Chapter 5, when we begin to generate our models and controllers.

Another example is the generation of a basic web-based interface to a model, referred to as **scaffolding**. We'll also look at scaffolding in Chapter 5, as we make a start on building our views.

ActionMailer

While not strictly part of the Web, email is a big part of our online experience, and Rails's integrated support for email is worth a mention. Web applications frequently make use of email for tasks like sending sign-up confirmations to new users and resetting a user's password.

`ActionMailer` is the Rails component that makes it easy to incorporate the sending and receiving of email into your application. `ActionMailer` is structured in a similar way to `ActionPack` in that it consists of controllers and actions with templates.

While the creation of emails, and the processing of incoming email, are complex tasks, `ActionMailer` hides these complexities and handles the tasks for you. This means that creating an outgoing email is simply a matter of supplying the subject, body, and recipients of the email using templates and a little Ruby code. Likewise, `ActionMailer` processes incoming email for you, providing you with a Ruby object that encapsulates the entire message in a way that's easy to access.

Adding email functionality to a web application is beyond the scope of this book, but you can read more about `ActionMailer` on the Ruby on Rails wiki.[5]

Testing and Debugging

As mentioned in Chapter 1, an automated testing framework is already built into Ruby on Rails. It also, rather helpfully, supplies a full stack trace for errors to assist with debugging.

Testing

A number of different types of testing are supported by Rails, including automated and integration testing.

[5] http://wiki.rubyonrails.com/rails/pages/ActionMailer/

Automated Testing

The concept of automated testing isn't new to the world of traditional software development, but it's fairly uncommon in web application development. While most Java-based web applications make use of comprehensive testing facilities, a large number of PHP and Perl web applications go live after only some manual tests have been performed (and sometimes without any testing at all!). Although performing automated tests is optional, developers may decide against this option for reasons ranging from the complexity of the task to time constraints.

We touched on this briefly in Chapter 1, but it's worth stressing again: the fact that comprehensive automated testing is built into Rails, and is dead easy to implement, means there's no longer a question about whether or not you should test your apps. *Just do it!*

The `generate` command that we introduced a moment ago can automatically create testing templates that you can use with your controllers, views, and models. (Note that Rails just assists you in doing your job, it's not replacing you—yet!)

The extent to which you want to implement automated testing is up to you. It may suit your needs to wait until something breaks, then write a test that proves the problem exists. Once you've fixed the problem so that the test no longer fails, you'll never again get a bug report for that particular problem.

If, on the other hand, you'd like to embrace automated testing completely, you can even write tests to ensure that a specific HTML tag exists at a precise position within a page's hierarchy.[6] Yes, automated tests *can* be that exact.

Integration Testing

Rails's testing capabilities also include **integration testing**.

Integration testing refers to the testing of several web site components in succession—typically, the order of the components resembles the path that a user would follow when using the application. You could, for example, construct an integration test that reconstructs the actions of a user clicking on a link, registering for a user

[6] The hierarchy referred to here is the Document Object Model (DOM), a W3C standard for describing the hierarchy of an (X)HTML page.

account, confirming the registration email you send, and visiting a page that's restricted to registered users.

We'll look at both automated testing and integration testing in more detail as we progress through the development of our application.

Debugging

When you're fixing problems, the first step is to identify the source of the problem. Like many languages, Rails assists this process by providing the developer (that's you!) with a full stack trace of the code. We mentioned earlier in the section called "Three Environments" that a stack trace is a list of all of the methods that were called up to the point at which an exception was raised. The list includes not only the name of each method, but also the classes those methods belong to, and the names of the files they reside within.

Using the information contained in the stack trace, you can go back to your code to determine the problem. There are several ways to tackle this, depending on the nature of the problem itself:

- If you have a rough idea of what the problem may be, and are able to isolate it to your application's model (either a particular class or aspect of your data), your best bet is to use the Rails console that we looked at earlier in this chapter. Type **console** from the **script** directory to launch the console. Once inside, you can load the particular model that you're interested in, and poke at it to reproduce and fix the problem.
- If the problem leans more towards something related to the user's browser or session, you can add a `debugger` statement around the spot at which the problem occurs. With this in place, you can reload the browser and step through your application's code using the ruby-debug tool to explore variable content or to execute Ruby statements manually.

We'll be covering all the gory details of debugging in Chapter 11.

Summary

In this chapter, we peeled back some of the layers that comprise the Ruby on Rails framework. By now you should have a good understanding of which parts of Rails perform particular roles in the context of an MVC architecture. We also discussed how a request that's made by a web browser is processed by a Rails application.

We looked at the different environments that Rails provides to address the different stages in the life cycle of an application, and we configured databases to support these environments. We also provided Rails with the necessary details to connect to our database.

We also had our first contact with real code, as we looked at the `ActiveRecord` models, `ActionController` controllers, and `ActionView` templates for our Shovell application. We explored the topics of the REST style of application architecture, code generation, testing, as well as debugging.

In the next chapter, we'll build on all this knowledge as we use the code generation tools to create actual models, controllers, and views for our Shovell application. It's going to be a big one!

What’s Next?

If you’ve enjoyed these chapters from *Simply Rails 2*, why not order yourself a copy?

You’ve only seen a small part of this easy-to-follow, practical and fun guide to Ruby on Rails for beginners. It covers all you need to get up and running, from installing Ruby and Rails to building and deploying a fully featured web application.

This book will teach you how to:

- Program with confidence in the Ruby language.
- Build and deploy a complete Rails web application.
- Exploit the new features available in Rails 2.
- Use Rails' Ajax features to create slick interfaces.
- Reap the benefits of a best-practice MVC architecture.
- Work with databases easily using ActiveRecord.
- implement RESTful development patterns and clean URLs
- Create a user authentication system.
- Use object oriented concepts like inheritance and polymorphism.
- Build a comprehensive automated testing suite for your application.
- Add plugins to easily enhance your application's functionality.
- Implement caching to alleviate database performance issues.
- Use migrations to manage your database schema without data loss.
- Achieve maximum code reuse with filters and helper functions.
- Debug your application using ruby-debug.
- Analyze your application's performance using the Rails logging infrastructure.
- Benchmark your application to determine performance bottlenecks.

Buy the full version now!

Index