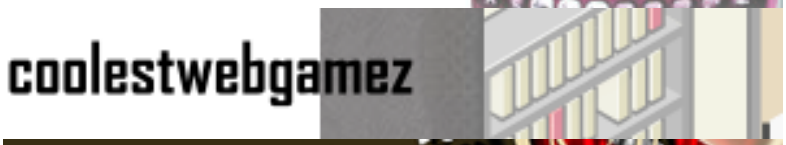
Download the coolest fonts for PC & MAC at a-font **[Click Here]**

Top 40 Wallpaper Websites on the Web **[Click Here]**

Latest Cell Phones reviewed plus video reviews **[Click Here]**

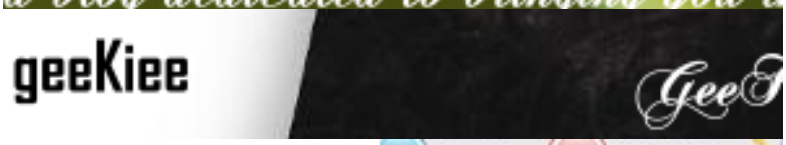


Coolest Online Web Flash Games, Addictive & Fun **[Click Here]**

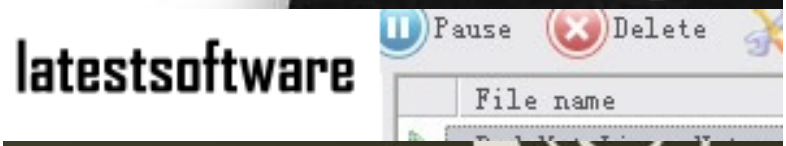
High resolution wallpapers, the best online.. **[Click Here]**

Free Ebooks & Magazines For download **[Click Here]**

Amazing Wallpapers to go with your Windows Vista **[Click Here]**



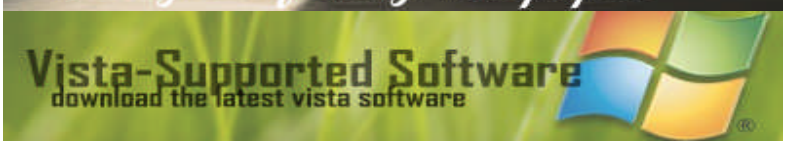
Cool Fun Tech News & Bookmarks **[Click Here]**



Latest Software Available For Download For Free **[Click Here]**

The Best Collection of Free Professional Website Templates for your website **[Click Here]**

A Collection of the Best Car Wallpapers Updated Often **[Click Here]**



Download Vista-Supported Software **[Click Here]**

THE EXPERT'S VOICE® IN OPEN SOURCE

Beginning Ruby on Rails E-Commerce

From Novice to Professional

Learn how to quickly develop next-generation online shops using Ruby on Rails

Christian Hellsten and Jarkko Laine

Apress®

Beginning Ruby on Rails E-Commerce

From Novice to Professional

Christian Hellsten and Jarkko Laine

Apress®

Beginning Ruby on Rails E-Commerce: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-736-1

ISBN-10 (pbk): 1-59059-736-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Keir Thomas
Technical Reviewer: Peter Marklund
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole Flores
Copy Editor: Marilyn Smith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Pat Christenson
Proofreader: Dan Shaw
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section.

Contents at a Glance

Contents

About the Authors

CHRISTIAN HELLSTEN is the founder of Aktagon Ltd., a provider of consulting services and custom Internet software development, and CTO of Sanda Interactive Ltd. He has worked on large-scale e-business projects as a consultant for PricewaterhouseCoopers Consulting and IBM Business Consulting Services. Christian's background is in J2EE, but he fell in love with Ruby on Rails at first sight, and has been using it professionally ever since to build web applications. When he is not changing the diapers of his two young daughters at his home in Finland, Christian enjoys researching new and better ways of building software.

JARKKO LAINE is the owner and CEO of O'Design, a Rails-based web design shop. He has been using Ruby on Rails since its public launch in 2004. He has contributed patches to the core developer team, and has also contributed to several Rails plugins. Jarkko has provided Rails consultancy for a number of organizations, from nonprofits to Fortune 500 companies. He has also taught Rails at the university level and delivers lectures about Rails around the world. Currently, he works on dotherightthing.com, a project that will bring people a whole new way to rate, follow, and discuss the social responsibility of companies. Jarkko is a sports junkie, so if he isn't sitting in front of his computer, he is probably running around forests or kicking a ball on the nearest field. He lives in Tampere, Finland, with his fiancée Maria and a growing list of pending household chores.

About the Technical Reviewer

PETER MARKLUND has extensive experience with and expertise in object orientation, web development, relational databases, and testing, and has been doing web development with Java and Tcl since 2000. He was one of the core developers of the OpenACS open source web framework. In late 2004, he was introduced to Ruby on Rails and has since helped develop an online community and a CRM system with Rails. Peter is currently working as a Ruby on Rails freelancer and is also helping organize events for the Ruby on Rails developer community in Stockholm. Peter has a personal blog at `http://marklunds.com`, where he shares Rails tips with other developers.

Acknowledgments

First of all, I would like to thank my family for allowing me to take on such a time-consuming project as this in my spare time. Secondly, I would like to thank everyone involved in this project, including Keir Thomas, Jarkko Laine, Peter Marklund, Beth Christmas, Marilyn Smith, and Kelly Winquist. Last, but not least, I would like to thank my parents, for buying me a Commodore VIC-20, back in the early 1980s.

Christian Hellsten

I am eternally grateful to the following people: Yukihiro "Matz" Matsumoto and David Heinemeier Hansson for bringing passion and joy back to programming; my ex-girlfriend—now fiancée—Maria, for putting up with the innumerable nights spent married to the computer; my parents, for telling me to believe in and pursue my dreams, even if it was just "fooling around with computers"; the whole team at Apress, for towing me back on track in the moments of despair; and finally, Philip and Alex, for igniting the spark.

Jarkko Laine

Introduction

Beginning Ruby on Rails E-Commerce is for people who want to learn how to build real-world professional web applications using Rails best practices. We put a specific emphasis on e-commerce by showing you how to build an online bookstore, including a shopping cart, catalog, forum, and other functionality. On the front-end, we guide you through important technologies like Ajax, syndication, tagging, and internationalization. On the back-end, we show you how to integrate with payment gateways, use ActiveRecord and the Ferret search engine, and many other techniques.

This book is also targeted at people who already have written an application or two using Rails, but who want to learn more about how test-driven development (TDD) can improve the quality of their code, and how to go beyond the standard test features built inside Rails.

We will guide you through all the phases of a professional e-commerce project, from concept to production deployment and maintenance. In the first chapters, we show you how to jump-start your project and build a good, solid foundation for it, using agile practices like TDD. In later chapters, we dig deeper into Ruby on Rails, covering common requirements, such as translating your application into multiple languages and debugging production problems.

Beginning Ruby on Rails E-Commerce is not intended to be a reference manual for Ruby on Rails. You can find many online resources and other books that provide a complete reference to the Ruby on Rails API and features, and these are mentioned throughout this book.

What Is Ruby on Rails?

Ruby on Rails (`http://rubyonrails.org`) is a web application framework written using the Ruby programming language. It was originally created by David Heinemeier Hansson, a Danish hacker, during the development of an online project collaboration tool called Basecamp.

As with most great things, Ruby on Rails started as an itch. Hansson was not happy with the available web application frameworks at the time, so he decided to write his own. In the design of Ruby on Rails, David emphasized a couple of things like convention over configuration, less software, and that programmer happiness ultimately leads to better productivity.

Ruby on Rails was first released to the public in July 2004. Since then, it has seen an explosive growth in popularity. It is loved because of its simplicity and power, which allow

you to solve problems faster and with less code than, as David said, "most frameworks spend doing XML sit-ups."

What Is Ruby?

Ruby (`http://ruby-lang.org`) is a dynamically typed programming language created by a Japanese Software Engineer called Yukihiro "Matz" Matsumoto in February 1993. Ruby is licensed under the GPL-like Ruby license and was released to the public in 1995, which is about one year later than Java. It is actively maintained by Matz and contributors from all over the world.

Unlike most other programming languages, Matz designed Ruby to increase programmer happiness, and to let programmers concentrate more on solving the task at hand than on language syntax. This is arguably the greatest strength of the Ruby programming language, when compared to other programming languages.

Ruby is a completely object-oriented language, unlike for instance Java, which has primitives. Everything in Ruby is an object, even `nil`. Ruby is also highly dynamic, allowing you to change classes and to introduce new methods at runtime. This allows the programmer to do things that aren't possible in languages like Java and C++.

CHAPTER 1

Project Setup and Proof of Concept

Ruby on Rails is highly suited for rapid prototyping; complex functionality can be implemented in hours or even minutes. This will come in handy, because the first thing George, our customer and the owner of Emporium, wants us to do is to implement a proof of concept. He needs to see with his own eyes that Ruby on Rails is not vaporware before he hands us the contract. We are happy to oblige.

In this book, we'll use a fictional bookstore project to make it easier for you to follow the process of implementing a web application from start to finish. In this chapter, we'll begin by introducing the Emporium project we will develop in this book. Then we will show you how to install Ruby on Rails and the software needed for implementing the first version of the Emporium application. Next, we'll provide a brief introduction to the Scrum lightweight project management process, which we use to manage the project team and requirements. Then we'll show you how to get started with Ruby on Rails by creating the Emporium application. Finally, we'll implement Emporium's About page as part of the proof of concept. This is a simple page that shows Emporium's contact details and will be implemented using code generation, a powerful built-in feature of Ruby on Rails.

Introducing the Emporium Project

We'll show you how to implement the project exactly as we would do in a real-world project.

One morning our coffee break is interrupted by a furious phone call. On the other end is George, the owner of Emporium, a hip bookstore in downtown Manhattan. George has just received the financial figures for the online sales of his shop, and he is not happy. "We're losing all our customers to Amazon." Something must be done.

Emporium's current online store is functional but rigid and slow, and the customers don't really like it. Sure, it was fine eight years ago, but now it's really starting to show its age. "Look at the shop at panic.com," says George, "you can drag things into the cart there. Why doesn't that work in my shop?" Sure, George, we got it. George also wants to empower the users more, with syndication of new content (you know, that RSS thingamagick) and forums. He has also heard that tagging is the concept du jour, something a self-respecting online store just can't live without.

While sitting at the back of his bookstore and spying customers, George has spotted a book called *Agile Web Development with Rails* being of interest to web hackers. While flipping through the book, he has discovered that Rails is like a breath of fresh air in the world of web applications. Now George wants to know if Rails would be a good fit for his website. "But it must do tagging," he reminds us, "and don't forget the drag thing!"

Since George is about the computer-savviest person in the whole store, the system must also be very easy to use even on the administration side. Turns out it also has to integrate with payment gateways, so George is able to bill his customers. And since George is worried about expenses, the system must not cost an arm and a leg (at maximum a leg). "Can you do it? Can Rails do it?" insists George. "Sure," I reply, just to get back to my coffee mug. But what's promised is promised, so it's time to get our hands dirty.

George is not the most organized person in the world, and like most of our customers he has no experience of IT projects. This would normally be a disaster for an IT project, but we have dealt with difficult customers and projects without clear requirements before.

In this book, you will not only learn how to build a working e-commerce site with Ruby on Rails, but we will also teach you techniques and best practices like test-driven development (TDD) that will improve the quality of your application.

Installing the Software

In this section, you will learn how to install the following software:

- Ruby, the interpreter for the Ruby programming language
- RubyGems, Ruby's standard package manager
- Ruby on Rails
- MySQL, an open source database
- Ruby MySQL driver

For our project, we will use Linux as our software development platform. Linux is highly suited as a software development platform, as there are many tools available that increase developer productivity. This section explains how to install all the required software on Ubuntu Linux. The instructions should also be valid, with minor exceptions, for other Linux distributions, since you will compile some of the software from source.

■ **Note** While we have tried to ensure that these instructions are valid, there's no guarantee that they will work without problems on your setup. If you encounter problems while following these instructions, use Google or another search engine to find a solution. You can also ask questions on the Rails IRC channel at `http://wiki.rubyonrails.com/rails/pages/IRC`.

INSTALLING RAILS ON WINDOWS OR MAC OS X

Throughout this book, we assume that Ubuntu is used as a development platform. However, you can also install and run Rails under Windows or Mac OS X.

Under Windows, you can either download and install everything separately or use Instant Rails, which is available for download at `http://instantrails.rubyforge.org`. Instant Rails is a preconfigured package containing everything you need for developing an application: Ruby, Ruby on Rails, Apache, and MySQL. It is perhaps the easiest way to get started with Ruby on Rails on Windows. However, we recommend that you install everything separately, since this allows you to learn more about the software that Ruby on Rails is built on and uses. Follow these simple instructions to manually install Ruby on Rails on Windows:

- Download and install the latest stable release of the One-Click Ruby Installer for Windows from `http://rubyinstaller.rubyforge.org`. This installer comes with RubyGems installed, so there's no need to install it separately.
- Install Ruby on Rails by executing `gem install rails --include-dependencies` in a command window.

Note that the installation of the native MySQL driver written in C is not covered by these instructions.

Installing Ruby on Rails on Mac OS X can also be done in two ways: by downloading everything separately or by using an all-in-one installer called Locomotive, which is available for download from `locomotive.sourceforge.net`. If you opt for installing everything separately, you can follow the installation instructions available at `http://hivelogic.com/articles/2005/12/01/ruby_rails_lighttpd_mysql_tiger`.

Ubuntu is a Debian-based and award-winning Linux distribution, which is suitable for both desktop and server use. Ubuntu comes with professional and community support and lives up to its promise, "Linux for human beings" by being easy to install and use. Ubuntu promises regular releases every six months and can be downloaded for free from `www.ubuntu.com`. For the instructions here, we assume that you have a fresh installation of Ubuntu.

Tip The latest installation instructions for most platforms can be found at `http://wiki.rubyonrails.com`, the Ruby on Rails wiki.

Installing Ruby

Your first step is to install the official Ruby interpreter, since Ruby on Rails is written in the Ruby programming language.

Log in to Ubuntu and open a console window. Check that Ruby is installed by typing `ruby --v` at the command prompt and then pressing Enter. To our disappointment, Ruby is not preinstalled on Ubuntu. You have at least two options for installing Ruby on Ubuntu:

- Use the `apt-get` command. This option is for Debian-based systems and requires only that you execute `apt-get install ruby`. You can also use the Synaptic Package Manager, a graphical front-end for `apt-get`, to install Ruby.
- Compile Ruby from source. This works on all Linux distributions and platforms, but requires a bit more knowledge.

Here, we will show you how to compile Ruby from source on Ubuntu, as this allows us to install the exact version we need, not the one provided by default by Ubuntu. Before continuing, you need to install some additional tools. Issue the following command:

```
$ sudo apt-get install build-essential zlib1g-dev
```

The `build-essential` package contains `make` and the Gnu Compiler Collection (GCC) package, which includes a C/C++ compiler that we will use to compile the source. The `zlib` package, also referred to as "A Massively Spiffy Yet Delicately Unobtrusive Compression Library," is needed by RubyGems, the standard package manager for Ruby.

Next, fire up your browser, go to `www.ruby-lang.org`, and click the Ruby link under Download. Choose to download the latest stable release that is compatible with Rails.

■ **Note** Before downloading Ruby, check which version is required by the Ruby on Rails version that you want to use by reading the online documentation found at `www.rubyonrails.org`. The documentation for version 1.1 of Ruby on Rails recommends version 1.8.4 of Ruby. Normally, you should install the latest stable release.

After you download Ruby, enter the following command to decompress it to a temporary directory (replacing the filename with the correct version):

```
$ tar zxvf ruby-version.tar.gz
```

Change to the directory where you extracted the source and execute the following commands to compile and install Ruby:

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

■ **Note** You need to belong to the `sudoers` list to execute the `sudo` command. The list of `sudoers` is defined in `/etc/sudoers`. Use the `visudo` command to add yourself to the list.

The `configure` script customizes the build for your system and allows you to specify parameters, which can be used to further customize the build.

The compilation is done by the `make` command according to the makefile generated by `configure`.

The last line, `make install`, requires superuser privileges, as it will install the compiled binaries to a shared directory.

If you have problems with the previous steps, check the `README` file for more detailed installation instructions and verify that all dependencies are installed. If you received an error message, try using a search engine to look for information about the error with a search query such as "install ruby" Ubuntu "*error message*" (replace *error message* with the error message you are getting).

If everything went well, Ruby is installed, and you can verify that Ruby works and has the correct version number:

```
$ ruby -v
```

```
ruby 1.8.4 (2005-12-24) [i686-linux]
```

Installing RubyGems

RubyGems, Ruby's standard package manager, provides a standard format for distributing Ruby applications and libraries, including Ruby on Rails itself. For example, later in the book, we will install the Ferret search engine and Globalize plugin with the help of RubyGems.

The software packages managed by RubyGems are referred to as *gems*, and can be downloaded either manually or by RubyGems itself from a central repository. The RubyGems installation files and the gems are hosted by RubyForge, the home of many open source Ruby projects.

Next, open `http://rubyforge.org` in your browser and search for the RubyGems project. Click the link to the RubyGems project's homepage, and then download the latest version.

After the download has completed, extract the contents of the package to a temporary directory. Remember to substitute the filename with the version you downloaded:

```
$ tar xzvf rubygems-version.tgz
```

Change the current directory to where the source was extracted, and execute the following command:

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

You should see the following result:

```
Successfully built RubyGem
Name: sources
Version: 0.0.1
File: sources-0.0.1.gem
```

Verify that the installation was successful by running the following command:

```
$ gem -v
```

You should see the version number printed out.

■ **Tip** Use the following command to upgrade the RubyGems installation itself later on: `gem update --system`. Use `gem update` to update all installed gems, such as Ruby on Rails and plugins. Note that you need to be careful and check that Ruby on Rails is compatible with all gems that get updated. For example, plugins and libraries might break your application if they are not compatible. Don't do this in a production environment without testing thoroughly to make sure that it works.

Installing Ruby on Rails

Now that RubyGems is installed, you can continue and install Ruby on Rails with the following command:

```
$ sudo gem install rails --include-dependencies
```

This tells RubyGems to install the latest version of Ruby on Rails and all its dependencies. It does this by downloading the packages, so you need to be connected to the Internet.

■ **Tip** You can install Ruby on Rails without access to the Internet by first downloading the Ruby on Rails gems and all the dependencies, and then executing `gem install path_to_gem`, replacing `path_to_gem` with the path and filename of the downloaded file.

Verify the installation by executing the following command:

```
$ rails -v
```

You should see the version number.

You can also run `gem list` to see a list of all gems that have been installed on your system, along with a brief description. Here is an example, with an abbreviated list of gems:

```
$ gem list
```

```
*** LOCAL GEMS ***

actionmailer (1.2.1)
    Service layer for easy email delivery and testing.

actionpack (1.12.1)
    Web-flow and rendering framework putting the VC in MVC.
```

Use the `about` command to get a more detailed view of your application's environment. The `about` script is located in the `scripts` directory of your Ruby on Rails application directory, which we will explain how to create in the "Creating the Emporium Application" section later in this chapter. Note that the version numbers shown in the following example will most likely be different on your system.

```
$ cd /home/george/projects/emporium
$ script/about
```

```
About your application's environment
Ruby version                 1.8.4 (i686-linux)
RubyGems version             0.8.11
Rails version                1.1.2
Active Record version        1.14.2
Action Pack version          1.12.2
Action Web Service version   1.1.2
Action Mailer version        1.2.1
Active Support version       1.3.1
Application root             /home/george/projects/emporium
Environment                  development
Database adapter             mysql
```

George has been monitoring our progress behind our backs. He is impressed by how simple it is to install Ruby on Rails, so he asks us if he can try it out at home on his Windows XP machine. We advise him to install Instant Rails and suggest that he read the Ruby on Rails documentation for more details, as each platform is a bit different.

Installing MySQL

Next, we ask George if he has a database we can use for storing the authors, books, and orders. He replies, "Sure, I have a database. Follow me to the office and I'll show you." To our horror, he fires up Microsoft Excel and proceeds to show us the orders from the previous eight years, all stored in a single spreadsheet. We try to keep a straight face and tell him that Ruby on Rails doesn't support spreadsheets, but it currently supports the following databases:

- Oracle
- IBM DB2
- MySQL
- PostgreSQL
- SQLite
- Microsoft SQL Server
- Firebird

Each of these databases has its own strengths and weaknesses. MySQL, which is what we will use in this book, is a good choice if you are looking for a fast and easy-to-use database.

MySQL is an open source database server that is developed and owned by MySQL AB, a Swedish company founded by David Axmark and Michael "Monty" Widenius. MySQL is used by many high-traffic websites, including `craigslist.com` and `digg.com`.

In this book, we use MySQL version 5, which supports advanced features like clustering, stored procedures, and triggers. This means that all of the examples and code have been tested with this version, and they might not work with earlier versions. You can either use a precompiled package or compile from source.

Go to the MySQL homepage at `www.mysql.com` and click the Developer Zone tab. Click Downloads and find and download the latest stable binary release of MySQL.

■ **Tip** You can also use `apt-get` on Ubuntu (`sudo apt-get install mysql-server`) to download MySQL, but you are not guaranteed to get the latest version.

We downloaded and installed the Linux binary package (not the RPM file that's offered for download), and then extracted the contents of the package (replace the filename with the name of the file you downloaded):

```
$ tar zxvf mysql-standard-version.tar.gz
```

To complete the installation, follow the instructions in the `INSTALL-BINARY` file, located in the root of the source directory.

■ **Tip** If you are new to MySQL, we highly recommend that you also install MySQL Query Browser and MySQL Administrator, which both can be downloaded from the MySQL Developer Zone page. MySQL Administrator, as its name implies, can be used for managing your MySQL sever. MySQL Query Browser allows you to run SQL queries and scripts from a graphical user interface. Both applications are available for Linux, Windows, and Mac OS X. If you're using OS X, a good application for both managing your databases and executing queries is CocoaMySQL (`www.theonline.org/cocoamysql/`).

Open a console window and execute the following command to start MySQL:

```
$ mysqld_safe --user=mysql &
```

The command starts the MySQL server in the background using the `mysql` user. Verify that you can connect to the database with the following command:

```
$ mysql -u root
```

You should see the following:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 5.0.19-standard

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Installing the MySQL Driver

Ruby on Rails needs a database driver to communicate with the MySQL server. Ruby on Rails comes with a pure Ruby MySQL driver, but we want to use the native C driver written by Tomita Masahiro, as it is considerably faster.

■ **Note** Each database requires a different driver, since there is no standard protocol. For more information about which databases are supported and how to get more information, refer to the Ruby on Rails wiki page on database drivers: `http://wiki.rubyonrails.com/rails/pages/DatabaseDrivers`.

First, you must install the MySQL development library before installing the native MySQL driver. On Ubuntu, you can find out which versions of the library are available by executing `apt-cache search libmysqlclient` (note that your system might have different versions of the library):

```
$ apt-cache search libmysqlclient
```

```
libmysqlclient10 - LGPL-licensed client library for MySQL databases
libmysqlclient10-dev - LGPL-licensed client development files for MySQL databases
libmysqlclient12 - mysql database client library
libmysqlclient12-dev - mysql database development files
libqt4-sql - Qt 4 SQL database module
libmysqlclient14 - mysql database client library
libmysqlclient14-dev - mysql database development files
```

Next, install the correct version with `apt-get`. Recall that we are using MySQL 5:

```
$ sudo apt-get install libmysqlclient14-dev
```

■ **Note** On Ubuntu, you should install `libmysqlclient14-dev` for MySQL 5, and `libmysqlclient12-dev` for MySQL 4.

Next, install the MySQL driver with the RubyGems `install` command:

```
$ sudo gem install mysql
```

Once again, RubyGems goes out on the Internet and downloads and installs the latest version of the MySQL driver. If everything goes well, you should see the following success message in the console:

```
Successfully installed mysql-2.7
```

If you get an error message, you probably forgot to install the MySQL development libraries or some other dependency.

Introducing Scrum

Scrum is an empirical and lightweight agile process, which we use to manage the project team and requirements. Scrum is mostly about common sense. Scrum embraces changes to requirements by keeping the time between software releases short. The biggest benefit of Scrum is perhaps the increase in productivity.

Scrum work is done in sprints. A *sprint* is the time period during which the next release of a system is being developed. It should be short—around two to four weeks, or even shorter. At the end of a sprint, you should normally have a working product, which can be shown to the customer or deployed into production. In this book, each chapter will describe the implementation of one sprint.

Most, if not all, software projects have a set of functional and nonfunctional requirements. In Scrum, these are analyzed and broken down into tasks that are documented with, for example, user stories or use cases.

■ **Note** A *user story* is a way of capturing the requirements for a project. User stories have a name and a description. The description is short—only a few sentences—and describes the requirement using the end user's language. User stories contribute to an active discussion between the customer and developers, because they are short and need clarification before implementation can start.

Scrum uses the product backlog and sprint backlog for keeping track of progress.

All tasks are written down and prioritized in the *product backlog*, which captures all the work left to be done in the project. For the Emporium project, we have identified a set of user stories and related tasks, which we have written down in the product backlog shown in Table 1-1. Note that the product backlog will evolve during the implementation of the Emporium application. New features will be added and old ones removed. We have also prioritized the items in the product backlog with the help of our customer and the product owner, George.

Table 1-1. *Initial Product Backlog Items for the Emporium Project*

Item	Description	Priority
1	Add author	Very high
2	Edit author	Very high
3	Delete author	Very high
4	List authors	Very high
5	View author	Very high
6	Add book	Very high
7	Edit book	Very high
8	Delete book	Very high
9	List books	Very high
10	View book	Very high
11	Upload book cover	Very high
12	About Emporium	Medium

Before starting work on the first real iteration, we need to identify the tasks that we are confident can be completed inside the sprint's time frame. In Scrum, sprint tasks are moved from the product backlog into the *sprint backlog*. The sprint backlog should contain only tasks that the team members are confident they can complete inside the selected time frame for the sprint. Table 1-2 shows the sprint backlog for the first sprint (sprint 0), which we'll implement in the next chapter.

Table 1-2. *Sprint Backlog for Sprint 0*

Item	Description	Priority
1	Add author	Very high
2	Edit author	Very high
3	Delete author	Very high
4	List authors	Very high
5	View author	Very high

Scrum is an agile and iterative process that we think most projects would benefit from using. Although Scrum is not suited for all projects and teams, we believe it is better than having no process at all. See `www.controlchaos.com` and `www.mountaingoatsoftware.com/scrum/` for more information about Scrum.

Creating the Emporium Application

We are now ready to start implementing the Emporium e-commerce site. We'll show George how fast we can create a Ruby on Rails application and implement one user story (About Emporium), which is enough for our proof of concept.

Creating the Skeleton Application

To create the Emporium application, run the `rails` command:

```
$ cd /home/george/projects
$ rails emporium
```

The `rails` command creates the directory structure and configuration for an empty Rails application in the current directory.

You can use the `tree` command to display the structure of the skeleton project the `rails` command created for you:

```
$ tree -L 1 emporium/
```

```
emporium/
|-- README
|-- Rakefile
|-- app
|-- components
|-- config
|-- db
|-- doc
|-- lib
|-- log
|-- public
|-- script
|-- test
`-- vendor
```

A brief description of the directory structure and files is provided in Table 1-3.

Table 1-3. *Directories and Files Located in the Root Directory*

Name	Description
`README`	Gives a brief introduction to Rails and how to get started
`Rakefile`	The application's build script, which is written in Ruby
`app`	Contains your application's code, including models, controllers, views and helpers
`components`	Empty by default; reusable code in the form of components should be placed here. Note that using components is generally considered a bad practice.
`config`	Holds your application's configuration files, including database configuration
`db`	Holds your ActiveRecord migrations and database schema files
`doc`	Empty by default; put the documentation for your application in this directory (use `rake appdoc` to generate the documentation for your controllers and models)
`lib`	Holds application-specific code that is reusable
`log`	Log files are written to this directory
`public`	Contains static files like HTML, CSS, and JavaScript files
`script`	Contains various scripts for starting and maintaining your Rails application
`test`	Holds your unit, integration, and functional tests and their fixtures
`vendor`	Where plugins are installed

Later, we will modify the skeleton application that Ruby on Rails created for us to fit our requirements. But first, we will show you how to create the Emporium database and how to configure Rails to use it.

Creating the Emporium Database

The database is where Emporium stores all its data. This includes authors, books, and order information. In a true agile fashion, we won't define the whole database schema immediately before starting the implementation. Instead, we will let the database schema evolve and update it in each chapter with the help of a powerful Ruby on Rails feature called *migrations*. Migrations will be introduced in Chapter 2, but in brief, migrations allow you to change your database schema incrementally. Each modification is implemented as a migration, which can then be applied to the database schema and even rolled back later.

■ **Note** We assume that you are familiar with MySQL. Sadly there isn't the space here to provide an introduction to SQL syntax. However, Apress publishes a number of excellent books that cover all aspects of MySQL use, including *Beginning MySQL Database Design and Optimization* (ISBN 1-59059-332-4) and *The Definitive Guide to MySQL, Third Edition* (ISBN 1-59059-535-1).

In fact, we will show you how to create three separate databases, one for each of the Ruby on Rails environments. Ruby on Rails builds on development best practices, which recommend that you use separate environments for development, testing, and production. Three databases for one application might seem like overkill at first, but the benefits are many. One is that each environment is dedicated to, and configured for, a specific task, as follows:

- The *development environment* is optimized for developer productivity. Ruby on Rails caches very little when in development mode. You can make a change to your application's code and see the change immediately, without redeployment or any compilation steps—just reload the page in your browser. This is perhaps the primary reason why Ruby on Rails is better suited for rapid application development than, for example, Java 2 Platform, Enterprise Edition (J2EE), which requires compilation and redeployment, slowing your development to a crawl.

- The *test environment* is optimized for running unit, integration, and functional tests. Each time you run a test, the test database is cleared of all data. Ruby on Rails can also be told to populate the database with test data before each test. This is done by using *test fixtures* (introduced in Chapter 2).

- The *production environment* is where every application should be deployed. This environment is optimized for performance, which means, for example, that classes are cached.

Environment-specific configuration related to the database is located in the `config/database.yml` file. Things related to code go into the `config/environment.rb` file. The environment-specific files are located in the `config/environments` directory.

■ **Caution** Always use a separate database for the test environment. If you, for example, use the same database for test and production, you will destroy your production data when running unit tests.

Creating the Development and Test Databases

You can use the MySQL command-line client to create the development and test databases. Connect as root to your MySQL server and execute the following commands:

```
$ mysql -uroot
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 5.0.19-standard
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql> create database emporium_development;
```

```
Query OK, 1 row affected (0.06 sec)
```

```
mysql> create database emporium_test;
```

```
Query OK, 1 row affected (0.00 sec)
```

This creates the two databases we need while developing the Emporium project. You can use the `show databases` command to display all databases on your server, including the ones you just created:

```
$ mysql -uroot
mysql> show databases;
```

```
+----------------------+
| Database             |
+----------------------+
| information_schema   |
| emporium_development |
| emporium_test        |
| mysql                |
| test                 |
+----------------------+
5 rows in set (0.28 sec)
```

As you might have noticed, we didn't create the production database. The production database is normally not used while developing and testing the application. If you want to, you can create it now, but we won't use it before we deploy Emporium to production in Chapter 12.

Setting Up the Database User

Next, create the MySQL user that will be used when connecting to the database environments. This is done by executing the following commands:

```
$ mysql -u root
mysql> grant all on emporium_development.* to \
  'emporium'@'localhost' identified by 'hacked';
```

```
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> grant all on emporium_test.* to \
  'emporium'@'localhost' identified by 'hacked';
```

```
Query OK, 0 rows affected (0.01 sec)
```

Note that we created only one user but granted access to both environments, with the `grant all` command. The first parameter, `emporium_development.*`, means we are giving all available privileges to the user. The second parameter, `'emporium'@'localhost'`, consists of two parts: the username and the IP address or address the user is allowed to connect from separated by `@`. The third parameter, `identified by 'hacked'`, assigns the password `hacked` to the user.

You can get a list of all users by executing the following SQL:

```
mysql> select host, user from mysql.user;
```

```
+-----------+----------+
| host      | user     |
+-----------+----------+
| localhost | emporium |
| localhost | root     |
+-----------+----------+
2 rows in set (0.00 sec)
```

■ **Caution** Don't give all available permissions to the MySQL user that will be used to connect to the production database. An application normally needs only select, insert, update, and delete privileges. For more information on the syntax of the grant command see `http://dev.mysql.com/doc/refman/5.0/en/grant.html`.

Configuring Ruby on Rails to Use the Database

The information Ruby on Rails needs for connecting to the database is located in a configuration file that is written in a lightweight markup language called YAML, an acronym for "YAML Ain't Markup Language." Ruby on Rails favors YAML over XML because YAML is both easier to read and write than XML.

The configuration file, `database.yml`, is located in the `db` folder and was created for you when you ran the `rails` command earlier in this chapter. The generated configuration template contains examples for MySQL, PostgreSQL, and SQLite. The MySQL configuration is enabled by default, and the other two database configurations are commented out.

Open the file and specify the database name and remove all of the text. Next, specify the database name, username, and password for the development and test environments, as shown in Listing 1-1.

Listing 1-1. *Emporium Database Configuration*

```
development:
  adapter: mysql
  database: emporium_development
  username: emporium
  password: hacked

test:
  adapter: mysql
  database: emporium_test
  username: emporium
  password: hacked
```

Save the configuration after you are finished editing it.

■ **Tip** The database configuration for each Rails environment is located in one file, `database.yml`. The runtime configuration for the development, test, and production environments are defined in separate configuration files. You can see the differences between the environments by comparing the `development.rb`, `test.rb`, and `production.rb` files, which can be found in the `config/environments` folder under the Rails application root. This is also where you should put the environment-specific configuration of your application.

Starting Emporium for the First Time

We are now ready to start up Ruby on Rails and Emporium for the first time, so we tell George to come over and have a look. We don't have to install any separate web servers, like Apache or LightTPD, while developing and testing Emporium. We can use the Ruby on Rails `script/server` command, which starts an instance of the WEBrick web server.

■ **Note** WEBrick is a Ruby library that allows you to start up a web server with only a few lines of code. WEBrick is suited only for development and testing, not production. In Chapter 12, we will show you how to set up and deploy the Emporium project to the LightTPD web server.

Next, execute `script/server` in the Emporium application directory to start WEBrick.

```
$ cd /home/george/projects/emporium
$ script/server
```

```
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-03-19 03:30:50] INFO  WEBrick 1.3.1
[2006-03-19 03:30:50] INFO  ruby 1.8.4 (2005-12-24) [i686-linux]
[2006-03-19 03:30:50] INFO  WEBrick::HTTPServer#start: pid=14732 port=3000
```

WEBrick is now running and configured to handle incoming requests on port 3000. Static content—like images, style sheets, and JavaScript files—are served by WEBrick from the `public` directory located under the application root directory. Requests for dynamic content are dispatched to and handled by Ruby on Rails.

Open `http://localhost:3000` in your browser to see the Emporium application in all its glory. You should see the default Welcome page shown in Figure 1-1. The most interesting thing on the page is the "About your application's environment" link, which, when clicked, takes you to a page that shows you some technical information about your application.

■ **Tip** You can start WEBrick and your application in different environments by using the environment parameter: `script/server -e test`. For example, the following will start your application's test environment and WEBrick in daemon mode, listening on port 80: `script/server -d -p 80 -e test`

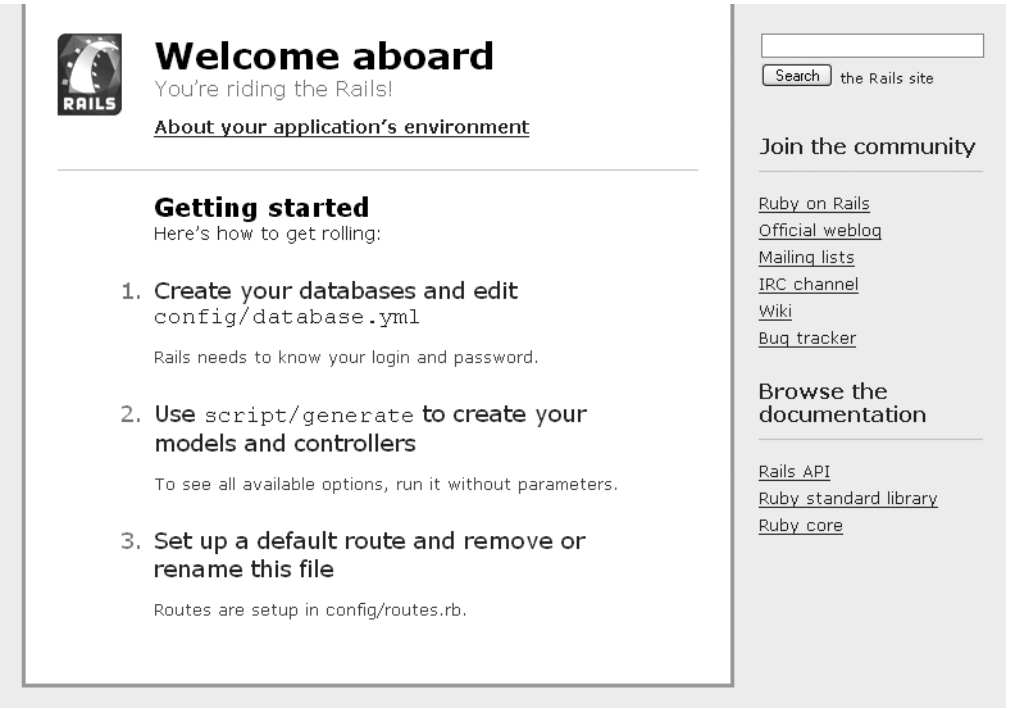


Figure 1-1. *The default Ruby on Rails Welcome page*

The code you see on the Welcome page is in the `index.html` file, which is located in the `public` directory under the application root. It was created by the `rails` command and should be deleted, so that it doesn't prevent the controller for the root context from being called:

```
$ rm /home/george/projects/emporium/public/index.html
```

You will get an error, `Recognition failed for "/"`, if you access `http://localhost:3000` again after deleting `index.html`. The error is thrown because there are no controller and action set up to handle the request.

We are now ready to start writing some code. But first, we'll introduce you to how requests are handled by the Rails framework.

How Does Ruby on Rails Work?

Ruby on Rails is built around the Model-View-Controller (MVC) pattern. MVC is a design pattern used for separating an application's data model, user interface, and control logic into three separate layers with minimal dependencies on each other:

- The *controller* is the component that receives the request from the browser and performs the user-specified action.
- The *model* is the data layer that is used, usually from a controller, to read, insert, update, and delete data stored, for example, in a relational database.
- The *view* is the representation of the page that the users see in their browser; usually, the model is shown.

■ **Tip** See Wikipedia's entry on MVC if you want to learn more about this pattern: `http://en.wikipedia.org/wiki/Model-view-controller`.

Figure 1-2 illustrates how a request from a browser is routed through the Ruby on Rails implementation of MVC. Ruby on Rails stores all MVC-related files in the `app` directory, which is located in the application root directory.

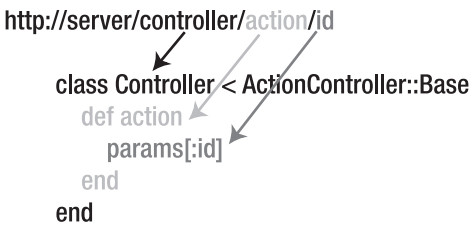


Figure 1-2. *A request routed through the Rails framework*

Implementing the About Emporium User Story

George has written the About Emporium user story on a piece of paper for us. He hands it over to us, and it reads as follows:

> *Emporium should have an About page where the contact details and a brief description of Emporium are shown to the user.*

We do not yet know exactly what text should be shown on the About page, but we'll first implement it, and then ask George again for more information when it is finished.

The requirement for the About Emporium user story is simply to display some description and the contact details for Emporium to the user. This is easy to implement and involves only two of the MVC layers: the controller and the view.

Running the Generate Script

First, we will jump-start the implementation of this requirement by using the `generate` script. The `generate` script can be used for quickly creating boilerplate code for controllers, models, and views or more complex functionality through third-party generators created by the Ruby on Rails community. The generated code usually requires modification to fit your requirements.

Run the `generate` script with the following parameters to create the `about` controller, `index` action, and related files.

```
$ cd /home/george/projects/emporium
$ script/generate controller about index
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/about
      exists  test/functional/
      create  app/controllers/about_controller.rb
      create  test/functional/about_controller_test.rb
      create  app/helpers/about_helper.rb
      create  app/views/about/index.rhtml
```

The `generate` script created a controller, view, helper, and a functional test for us. The controller has one action, `index`, which is called by default if no action is specified.

Next, open `http://localhost:3000/about` in your browser. You should see the About page we just created, as shown in Figure 1-3.

Figure 1-3. *The About page generated by Rails*

The "About#index" is computer-generated text, and the layout is ugly, so in the next section, we will spend some time modifying the code to fit our requirements.

As we told you earlier, the `generate` script creates a functional test and a helper class. In our case, we will not use the functional test or the helper. In later chapters, we will teach you how to use the test-driven development technique to write not only functional tests, but also unit and integration tests.

■ **Tip** Helpers are useful for keeping your views clean and readable. Views shouldn't contain complex logic and algorithms. Instead, you should refactor your view code and move the complex logic to a helper class. The methods in this class are automatically made available to the view. Using helpers will make your code easier to read and more maintainable.

Modifying the Generated View

Next, we tell George to come over and have a look at the About page. He asks us why the hell we have put the text "Find me in app/views/about/index.rhtml" on the page. We explain to him that the page, or *view* as it is also called, was generated by Ruby on Rails, and that we can change it. He scribbles down something on a paper, which looks like a foreign mailing address, and gives it to us.

A view is where you put the code for the presentation layer that generates, for example, HTML or XML. Views are written in a template language called Embedded Ruby (ERB), which allows you to write Ruby code directly in the view. Here is an example of a view written in ERB that prints out the text "This is embedded Ruby code" to the browser.

```
<%# This is a comment and is not shown to the user %>
<% text = "This is embedded Ruby code" %>
<%= text %>
```

ERB code follows the syntax `<% Ruby code %>`, and `<%= Ruby expression %>` is used for printing out the value of an object to the browser. Comments are formatted as `<%# comment %>`.

ERB also allows you to prevent HTML injection by escaping the following special characters in content entered by users: &, <, >, and ". The following line outputs the text `&lt;&gt;` to the browser, instead of &<>, by using the `h` method, which is short for `html_escape`.

```
<%= h('&<>') %>
```

The view we are going to change is located in the `app/views` directory. This is the root directory for all views. Each view is stored in a subdirectory named after the controller. For example, if the path to your controller is `app/controllers/about_controller.rb`, the path to that controller's views is `app/views/about`.

Open `app/views/about/index.rhtml` in an editor and change it as follows:

```
<p>Online bookstore located in downtown Manhattan, New York</p>
<h2>Mailing Address</h2>
<address>
Emporium<br/>
P.O. Box 0000<br/>
Grand Cayman, Cayman Islands<br/>
</address>
```

After saving the changes, go back to the browser and click the reload button. You should see the page shown in Figure 1-4.

Online bookstore located in downtown Manhattan, New York

Mailing Address

Emporium
P.O. Box 0000
Grand Cayman, Cayman Islands

Figure 1-4. *The modified About page*

Creating the Layout

George is a bit happier now, but he says that the page is not as nice looking as the current Emporium website, which he tells us was designed eight and a half years ago by his then ten-year-old nephew. He agrees that Emporium needs a new site design, but he tells us that he just sent some money to his starving sister on the Cayman Islands. So we decide to implement a design that can be improved later, because it will take a month or two before George can afford a professional designer.

Layouts are used in Ruby on Rails for surrounding the content of your pages with a header and footer. Figure 1-5 illustrates the concept of Rails layouts and views. The example shows a typical page consisting of a header, content, and footer section.

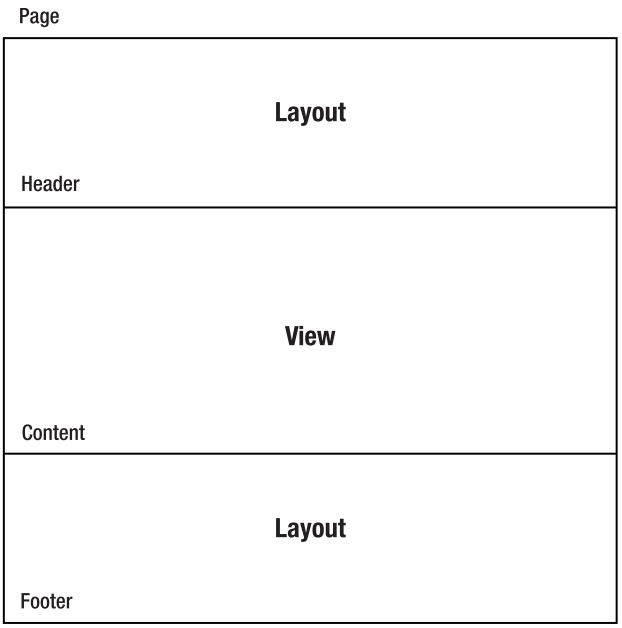


Figure 1-5. *Layouts and views*

The same result can also be achieved by inserting the same header and footer code in all views, but this goes against the Don't Repeat Yourself (DRY) principle, which states that you should avoid code duplication in all parts of your code.

Consider the following HTML for a page generated by Rails.

```
<html>
  <head>
    <title>Emporium</title>
  </head>
  <body>
    <!--Content start-->
    Page content
    <!--Content end-->
  </body>
</html>
```

All content above the text `<!--Content start-->` and below `<!--Content end-->` comes from the following layout file. The view contains the text "Page content."

```
<html>
  <head>
    <title>Emporium</title>
  </head>
  <body>
    <!--Content start-->
    <%= yield %>
    <!--Content end-->
  </body>
</html>
```

Listing 1-2 shows a very minimalist layout, which is enough for the proof of concept. Enter it in an editor and save the contents in `app/views/layouts/application.rhtml`.

Listing 1-2. *Emporium's First Layout*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title><%= @page_title || 'Emporium' %></title>
    <%= stylesheet_link_tag "style" %>
  </head>
  <body>
    <%= "<h1>#{@page_title}</h1>" if @page_title %>
    <%= yield %>
  </body>
</html>
```

The layout contains four Ruby expressions that are all evaluated when the page is rendered. The first expression will allow us to set the page title by defining an instance variable in our controllers:

```
<%= @page_title || 'Emporium' %>
```

The default title, "Emporium," is used if we don't specify the `@page_title` variable in the controller.

The second expression includes the Emporium style sheet, which we'll create in the next section.

■ **Tip** The `stylesheet_link_tag` automatically appends the last modification date to the URL of the style sheet file, for example, `/stylesheets/style.css?1150321221`. Appending the modification date will make the browser download the style sheet again, when it is updated, instead of taking a stale one from the browser cache. The same logic is used for the `javascript_include_tag`.

The third expression allows us to use the title of the page as the page heading:

```
<%= "<h1>@page_title</h1>" if @page_title %>
```

If the instance variable `@page_title` has not been defined in the controller, then nothing is shown.

The last expression inserts the view part of the page by a call to the `yield` method.

Creating a Style Sheet

The About page looks a bit plain. It's using the browser's default font and font sizes. To make the page look nicer, we'll tell Emporium to use a style sheet. *Style sheets* separate presentation from content and allow you to define, for example, the font and colors of HTML elements. The biggest benefit of using style sheets is that it allows us to separate content from presentation. This allows us to change the whole look and feel of our site by modifying only the style sheet.

■ **Tip** For more information about style sheets, see the World Wide Web Consortium's page on Cascading Style Sheets: `www.w3.org/Style/CSS/`.

The standard way of including style sheets in Rails is through the `stylesheet_link_tag`, as shown in Listing 1-3.

Listing 1-3. *The Initial Version of Emporium's Style Sheet*

```
body { background-color: #fff; color: #333; }

body, p, ol, ul, td {
  font-family: verdana, arial, helvetica, sans-serif;
  font-size:   13px;
  line-height: 18px;
}

pre {
  background-color: #eee;
  padding: 10px;
  font-size: 11px;
}

a { color: #000; }
a:visited { color: #666; }
a:hover { color: #fff; background-color:#000; }
```

Note that we have included only an excerpt from the style sheet; the complete style sheet can be downloaded from the Source Code section of the Apress website (`www.apress.com`). Save the style sheet in `public/stylesheets/style.css`.

Using Multiple Layouts

An application can have many layouts; for example, it may have one for normal pages and one for pop-up windows. You can tell Ruby on Rails to use a specific layout in many ways, of which three are shown here:

```
class EternalLifeController < ApplicationController
  # Option 1
  layout 'default'
  # Option 2
  layout :determine_layout

  def index
    # Uses app/views/layouts/default.rhtml
  end

  def popup
    # Uses app/views/layouts/popup.rhtml
    # Option 3
    render :layout => 'popup'
  end
```

```
  def determine_layout
    if params[:id].nil?
      return "fancy_layout"
    else
      return "default"
    end
  end
end
```

Options 1 and 2 are self-explanatory. Option 2 uses a method that decides which layout to use based on some runtime information; in this case, it checks if the `id` parameter is null and uses the `fancy_layout` in that case.

The easiest way of changing the layout is by creating a file named `application.rhtml`. This is the default layout file and will be used by Rails without the need for manually specifying the layout.

Modifying the Generated Controller

The last thing we need to do to complete the proof of concept is to modify the controller and action. The controller is where the main logic of your application is located. Each controller has one or more actions that execute the business logic.

The Ruby on Rails `generate` script already created a controller for us in `app/controllers/`. Change it as follows. Note that we set the page title to "About Emporium."

```
class AboutController < ApplicationController
  def index
    @page_title = 'About Emporium'
  end
end
```

George is still standing behind our backs. He yells, "I've been standing here for 15 minutes, guys. I don't have the whole day! Where's my proof of concept?" We again reload the page in the browser, and he can finally see his proof of concept—a working About page showing a brief description and Emporium's address, as shown in Figure 1-6.

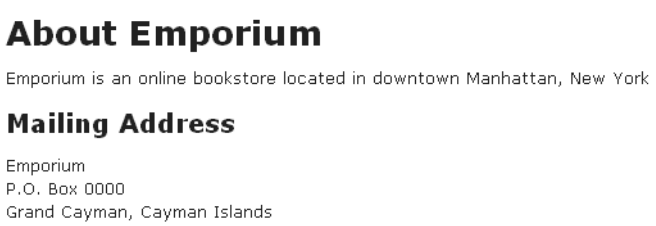


Figure 1-6. *The completed proof of concept*

Summary

In this chapter, we showed you how to implement a simple proof of concept for the Emporium project. We first explained how to install Ruby, Ruby on Rails, MySQL, and the MySQL driver. Then we showed you how to create a Ruby on Rails project skeleton using the `rails` command. Next, we introduced you to controllers, views, and layouts, which we used for implementing the About Emporium user story.

In the next chapter, we'll implement the user stories related to author management and introduce you to concepts like Test Driven Development (TDD) and ActiveRecord.

CHAPTER 2

Author Management

In this chapter, we start building the online bookstore for George for real. We first explain test-driven development (TDD), why we should use it, and how we can use it with Rails. We also describe the testing schemes that exist in Rails. After the introduction to TDD, we dive into our first sprint, implementing the author management system for our application. This is the first part of the administration interface to our bookstore application, to allow handling creating, reading, updating, and deleting of book authors (often referred to as CRUD capabilities). After completing this chapter, you should be able to use TDD to make a simple application in Rails.

Using Test-Driven Development

Software testing means using quality-assurance metrics to make sure the software works as it should. There are basically two ways to test software: manual testing, with dedicated test engineers banging the heck out of the software, and automated testing. While manual testing is needed in parts of the software that are hard to test programmatically (mainly the user interface), the bulk of testing can be done automatically. Automated testing is much faster and repeatable, and thus more systematic and less error-prone than testing everything manually.

While Ruby on Rails makes it easy to write automated tests for your application, we'll take this one step further and write our application test-driven. TDD starts from so-called *user stories*. One user story could be "George logs in to the system and adds a new author." After we have a user story, we have enough information to write a test. In our example, we could test that the login works and a new author is really created when George uses the application.

The real meat of TDD is that the actual code is written only after you have created the test for a user story, using the following process (from `http://wiki.marklunds.com/index.php?title=Test_Driven_Development_with_Ruby#What_is_Test_Driven_Development_.28TDD.29.3F`):

1. Write a test that specifies a bit of functionality.
2. Ensure the test fails. (You haven't built the functionality yet!)
3. Write only the code necessary to make the test pass.
4. Refactor the code, ensuring that it has the simplest design possible for the functionality built to date.

You read that correctly. No real code is written until you have a failing test in place to test the story at hand.

So what do we get from developing our application using TDD? First and foremost, when our tests pass without errors or failures, we can be certain that our application works just as we want it to—given that we wrote our tests well. Instead of throwing our application over the wall to George, crossing our fingers, and hoping that it won't explode, we can sleep well at night, knowing everything will go fine during that first demo.

But there is also another very important reason for TDD. As we try to be as agile as possible, we also want to obey one of the golden rules of Rails and 37signals: "Write less software." When we create the tests first and implement the functionality after that, writing the code has a clear goal: to implement the user story and make the tests pass. It is a lot easier to write only the code needed when you have a clear target than it is to first write the code and then start writing tests afterwards to test functionality you've already implemented.

Testing in Rails

As of version 1.1, Ruby on Rails has three different testing schemes built in: unit testing, functional testing, and integration testing. All of the Rails tests use (either directly or indirectly) the `Test::Unit` Ruby library.

In `Test::Unit`, tests are built on the notion of *assertions*. Assertions are methods that test the output of their arguments. The mother of all assertions methods, `assert`, for example, tests that its argument's code returns `true`. If the return value of the assertion argument is `false` or `nil`, the assertion has failed, and the failure will be reported.

Unit Testing

Unit testing is used in Rails to test business logic objects, represented by ActiveRecord models. (ActiveRecord is the object-relational mapping system in Rails, and it is described in the next section in this chapter.) Common tests for ActiveRecord models check that, for example, all validations work as they should and that all the methods you have written yourself work as intended.

For example, the following code tests that the validation of a new `Person` object does not pass unless the object has both the first and last name specified. It also checks that the method `Person#age` calculates the age correctly from the date of birth of a person. For that, we use another useful assertion method, `assert_equal`, which makes sure that its two arguments are equal.

```
class PersonTest < Test::Unit::TestCase
  def test_validation
    p = Person.new(:first => 'George', :last => nil)
    assert !p.valid?
    p.last = 'Pork'
    assert p.valid?
  end
```

```
  def test_age
    p = Person.create(:first => 'George', :last => 'Pork',
                      :dob => (Date.today - 35.years))
    assert_equal 35, p.age
  end
end
```

Functional Testing

Functional tests in Rails test single controllers and can be used to test that simple user stories work correctly. While unit tests in Rails focus on single ActiveRecord classes, functional tests focus on controller actions, which are called with the `get` and `post` helpers, just as they would be called by real HTTP requests.

Functional tests have a bunch of Rails-specific assertions for testing the response that results from the action. We will use functional tests extensively later in this chapter to implement the author management functionality for the Emporium project.

Integration Testing

Integration testing is a newcomer in Rails 1.1. Unlike functional tests, integration tests can span multiple controllers and be used to test Rails *routing*. Therefore, integration tests can be used to test complete user stories, ranging from signing in, to putting things into the shopping cart, to checking out. We will discuss integration testing in more detail in Chapter 3.

■ **Tip** Routing is a built-in system in Rails to map URLs to controllers and actions and vice versa. The default route in Rails is `:controller/:action/:id`, which means that, for example, the URL `http://localhost:3000/books/show/1` would be routed to the `BooksController`'s action `show`, and the action would be to receive an `id` parameter with a value of `1`. While the default routing is intuitive and easy to use, custom routing can be used to construct very flexible URL schemes.

Creating the ActiveRecord Model

ActiveRecord is the object-relational mapping system in Rails, thus denoting the *Model* in the Model-View-Controller (MVC) pattern. The job of the model part in the MVC paradigm is to take care of handling the data storage of the application. However, ActiveRecord is much more than simply a Ruby library for creating and executing SQL queries. It automatically maps database tables to classes in a Rails application, creates public methods for all database fields, and adds a load of useful methods for accessing the data in the database.

As noted earlier, we will be completing sprint 0 of our project in this chapter. This sprint involves five stories related to managing authors in the Emporium application. So first, we need a way to store our authors.

Using ActiveRecord Migrations

We could create the `authors` table by using the direct SQL commands. However, Rails has a great database-agnostic system for keeping up with the changes in the database—*migrations*.

When using migrations, the description of the database is written in pure Ruby. The different database adapters in Rails then interpret these commands to database-specific SQL commands. Another big advantage of using migrations is that you can keep track of the changes in the database schema and traverse back and forth between different schema versions. You can also deploy the migrations to multiple database servers simultaneously—something that will be very valuable when your server stack gets bigger.

Creating the Schema

Because our database doesn't contain anything yet, we start by creating the initial schema. As of Rails 1.1, creating a new ActiveRecord model will also create a new migration file for the corresponding table, so we use the `script/generate` command to create both the `Author` class file and the database migration for the `authors` table:

```
$ script/generate model Author
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/author.rb
      create  test/unit/author_test.rb
      create  test/fixtures/authors.yml
      create  db/migrate
      create  db/migrate/001_create_authors.rb
```

Not only did this single command create a model class file for the `Author` model, but it also created a skeleton for unit tests for that class and a new migration file, `001_create_authors.rb`. The number `001` at the beginning of the filename means that we're talking about the first migration for this application.

Editing the Migration File

The next step is to open the migration file in a text editor and edit it so that the database will look how we want it to look. In this case, it means adding two fields to the table skeleton:

```
class CreateAuthors < ActiveRecord::Migration
  def self.up
    create_table :authors do |t|
      t.column :first_name, :string
      t.column :last_name, :string
    end
  end
```

```
  def self.down
    drop_table :authors
  end
end
```

Rails migrations use a special domain-specific language (DSL) written in Ruby. This way, the migration code is database-agnostic and can be used to deploy the application to different database platforms.

Every migration should implement the methods `self.up` and `self.down`, which are created automatically when a new migration is generated by running either the `generate model` or the explicit `generate migration` command. Code inside `self.up` is run when the database is migrated to a higher version, and the code in `self.down` is run when the database is migrated back to an earlier version. In our first migration, we create a new table called `authors` and add columns for both the first and last names of an author. In the `self.down` part, we drop the `authors` table, just in case we someday would want to go back to ground zero.

■ **Note** Even though every table used by ActiveRecord (except for join tables) should have a primary key field called `id`, we didn't specify the creation of that column in our migration. This is because Rails will automatically create that field for every table, unless you explicitly tell it not to by using the `:id => false` option with `create_table`.

The commands in the migrations DSL mostly match the corresponding SQL statements. The following are the most commonly used commands. For complete documentation, see the Rails API documentation for migrations at `http://api.rubyonrails.org/classes/ActiveRecord/Migration.html`.

- `create_table(table_name, options)`: Creates a new table `table_name`. If it is given a block parameter (as in the `CreateAuthors` migration shown in the previous section), the commands inside the block will be executed inside the scope of this table.
- `column(column_name, type, options)`: Creates a new column `column_name` of type `type` in the scope of enclosing `create_table` block.
- `add_column(table_name, column_name, type, options)`: Equivalent to `column`, with the distinction that it is not used inside a `create_table` block, and thus needs the name of the table as an argument.
- `drop_table(table_name)`: Drops the table `table_name` from the database.
- `remove_column(table_name, column_name)`: Removes column `column_name` from table `table_name`.

Running the First Migration

After we are finished editing the migration file, we can run the first migration, as follows:

```
$ rake db:migrate
```

```
(in /home/george/projects/emporium)
== CreateAuthors: migrating ===================================================
-- create_table(:authors)
   -> 0.1776s
== CreateAuthors: migrated (0.1778s) ==========================================
```

Rails uses the development environment by default, so our development database is now updated to match our migration file. If you want to do the migration to some database other than the development database, you can do so by specifying the `RAILS_ENV` environment variable before the `rake` command:

```
$ RAILS_ENV=production rake db:migrate
```

So what really happened when our migration script was run? First of all, we got a new table called `authors`. But as this was the first migration, Rails also automatically created a table called `schema_info`. The table has only one field, `version`, and there should be only one row in that table at any given moment. The value of the row tells the current migration version of the database schema. The migration scripts use this information to determine which migrations need to be run to get everything up-to-date.

Last, the migration created a file called `db/schema.rb`, which is in the same format as the migration files, and always describes the *current* state of the whole database schema. After the file was created, future migrations will automatically keep it up-to-date. Therefore, you should never edit it by hand.

RAKE

Rake (`http://rake.rubyforge.org/`) is a build language and tool similar to `make` (`www.gnu.org/software/make/`) and `ant` (`http://ant.apache.org/`). It is written in Ruby and sports its own DSL for handling the maintenance of a Ruby application. Rails uses Rake extensively for many kinds of tasks. The following are some of the most popular Rake tasks used in Rails. For a complete list, run `rake -T` in the root of your Rails application directory.

- `rake`: Running `rake` without any parameters will rebuild the test database according to the migrations, and run all unit, functional, and integration tests found in the `test` directory.
- `rake db:migrate`: Updates the database of the current environment to the latest version. You can specify the target version by appending `VERSION=x` after the command.
- `rake db:sessions:create`: Creates a table for storing user session data in the database. Rails automatically assigns a session cookie for each user and uses it to track the user. The session mechanism is very useful for tasks like user authentication, as you'll see in Chapter 8.
- `rake db:sessions:clear`: Purges the `sessions` table. It is a good idea to schedule this command to run on regular intervals to keep the table size from growing rapidly. Every new visitor to the application will result in a new database row in the `sessions` table.
- `rake log:clear`: Truncates the log files of your Rails application's log directory. Just like session data, the log files can get massive over time, so it's a good idea to clear them every once in a while.
- `rake rails:freeze:gems`: Locks your Rails application to the latest version of Rails gems you have installed on your system. Without running this command (or `rake rails:freeze:edge`), your application will "float" on the latest gem version, which might lead to problems if there are changes in Rails code that break backward-compatibility.
- `rake rails:freeze:edge`: Similar to `rake rails:freeze:gems`, with the distinction that it locks the Rails code to the latest (possibly unstable) code in the Rails Subversion source code repository. The Rails code is copied to the `vendor/rails` directory in your application tree.
- `rake rails:unfreeze`: Breaks the connection between the application and Rails version that was created by either of the freeze tasks just described.
- `rake stats`: Outputs useful statistics about your application, including lines of code and the code to test ratio.

Rake is a very powerful tool that you can use to automate many of the repetitive and tedious maintenance tasks in your application. If you want to know more about Rake, Martin Fowler has written an excellent tutorial called "Using the Rake Build Language" (`http://martinfowler.com/articles/rake.html`).

Running Unit Tests

Since ActiveRecord creates methods for each field in the database table for a given ActiveRecord model, `Author` objects now automatically respond to the methods `first_name` and `last_name`. However, it would be nice to get the whole name of an author with a single method call. Let's first create a unit test that checks that our method works correctly.

Open `test/unit/author_test.rb` and replace the `test_truth` method with the `test_name` method:

```
require File.dirname(__FILE__) + '/../test_helper'

class AuthorTest < Test::Unit::TestCase
  fixtures :authors

  def test_name
    author = Author.create(:first_name => 'Joel',
                           :last_name => 'Spolsky')
    assert_equal 'Joel Spolsky', author.name
  end
end
```

Running `rake`, we notice that the test will result in an error, because the `Author` class doesn't have a method called `name` yet. Let's fix that by opening `app/models/author.rb` and implementing the method.

```
class Author < ActiveRecord::Base
  def name
    "#{first_name} #{last_name}"
  end
end
```

The method returns a string containing the return values of the `first_name` and `last_name` methods separated by a space. Running `rake` again tells us that everything is in order.

```
$ rake
```

```
(in /home/george/projects/emporium)
/usr/local/bin/ruby -Ilib:test
"/usr/local/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/unit/author_test.rb"
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.
Finished in 0.062438 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
/usr/local/bin/ruby -Ilib:test
"/usr/local/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/functional/about_controller_test.rb"
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.
Finished in 0.021772 seconds.

1 tests, 1 assertions, 0 failures, 0 errors

/usr/local/bin/ruby -Ilib:test "/usr/local/lib/ruby/gems/1.8/gems/
rake-0.7.1/lib/rake/rake_test_loader.rb"
```

Creating the Controller

As explained in Chapter 1, a controller is the central part of an application that takes care of receiving the user request, modifying data through the model part, and finally either rendering a view template or redirecting the user to another URL. Now that we have the ActiveRecord model for authors in place, we need a controller that implements the administration interface for authors. From that last sentence, we can already pick a good name for the controller: `admin/author`.

You create a controller by using the same `script/generate` command that you use to create a model. You can also give the `generate controller` command names of the actions you want the controller to implement—in this case, `new`, `create`, `edit`, `update`, `destroy`, `show`, and `index`.

```
$ script/generate controller 'admin/author' new create edit update ➥
                  destroy show index
```

```
      create  app/controllers/admin
      create  app/helpers/admin
      create  app/views/admin/author
      create  test/functional/admin
      create  app/controllers/admin/author_controller.rb
      create  test/functional/admin/author_controller_test.rb
      create  app/helpers/admin/author_helper.rb
      create  app/views/admin/author/new.rhtml
      create  app/views/admin/author/create.rhtml
      create  app/views/admin/author/edit.rhtml
      create  app/views/admin/author/update.rhtml
      create  app/views/admin/author/destroy.rhtml
      create  app/views/admin/author/show.rhtml
      create  app/views/admin/author/index.rhtml
```

By using `admin/author` instead of just `author` as the controller name, we put the `author` controller inside an `admin` subdirectory. The new `AuthorController` class is also set inside a module called `Admin`. This way, we can later implement other administrative controllers under the same module and make them share common code such as access control.

Just as for models, the `generate` command created a controller stub and empty views for all actions for us. It also created a functional test file for our controller.

■ **Note** If you haven't already done so, now would be a good time to put your code under source control. We prefer Subversion, as it's the norm in the Rails world, but the most important thing is that you use some kind of source control management. You wouldn't want to tell George that you lost a whole day's worth of code, would you?

Implementing the User Stories

"Test-driven development sure must be great," says George, "but I want to see something real, plenty soon." He is also not happy to hear that the sprints normally take more than a week to implement. We reassure him that using Rails, the sprints really live up to their names, and that we should have the first sprint ready before he shuts his doors for the day. A bold promise, so let's move on.

As you saw in our sprint backlog in Chapter 1 (Table 1-2), we need to implement five user stories in this sprint:

- *Add author*: When George hears about a new cool author, he must be able to get that author on his site. A click, typing the author details in a form, submitting the form, and the info should be there.
- *List authors*: The author administration interface needs a homepage, which also works as a list of all authors.
- *View author*: When the system evolves and more information about authors is added, there must be pages showing the details of an individual author.
- *Edit author*: Sometimes, author details change. Some people get married; others convert to a cargo cult and change their name to Ilyushin. Either way, nothing is perpetual. Changing author details needs to be as easy for George as adding new authors.
- *Delete author*: During his vacation, George's summer aides sometimes add totally bogus stuff into the system. They think it's funny. George doesn't, and neither do many of his customers. So it's important that authors can be deleted from the system, too.

We will proceed story by story, first writing a functional test for the story and then making the test pass by implementing the feature.

Adding an Author

The first user story to implement in this sprint is adding a new author.

Adding a Test Case

First, we just want to request the page with the form for adding the author and make sure that we're handed the correct template. We add a test case for it in `test/functional/admin/author_controller_test.rb`, which is the functional test file for our controller. When we open the file, we can see that there is a dummy `test_truth` test method in the file. As the comment above the method suggests, we remove it and replace it with our own test, `test_new`, shown in Listing 2-1.

Listing 2-1. *test/functional/admin/author_controller_test.rb*

```
require File.dirname(__FILE__) + '/../../test_helper'
require 'admin/author_controller'

# Re-raise errors caught by the controller.
class Admin::AuthorController; def rescue_action(e) raise e end; end

class Admin::AuthorControllerTest < Test::Unit::TestCase
  def setup
    @controller = Admin::AuthorController.new
    @request    = ActionController::TestRequest.new
    @response   = ActionController::TestResponse.new
  end

  def test_new
    get :new
    assert_template 'admin/author/new'
    assert_tag 'h1', :content => 'Create new author'
    assert_tag 'form', :attributes => {:action => '/admin/author/create'}
  end
end
```

In the test, we first use the `get` test helper method to call the `new` action. Then we use the Rails-specific `assert_template` method to test that we were indeed rendered the correct view file. Last, we check that the rendered template indeed looks as it should. We do this by testing that there are two HTML elements present: a level 1 heading with the content "Create new author," and a form element pointing to `/admin/author/create`.

Now we can go back to the command line and run our test case. We don't want to execute all the tests this time, so we just run the test file we're working on right now.

```
$ ruby test/functional/admin/author_controller_test.rb
```

```
Loaded suite test/functional/admin/author_controller_test
Started
F
Finished in 0.052617 seconds.

  1) Failure:
test_new(Admin::AuthorControllerTest)
[test/functional/admin/author_controller_test.rb:17]:
expected tag, but no tag found matching {:content=>
'Create new author', :tag=>'h1'} in:
"<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\"
        \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">
<html xmlns=\"http://www.w3.org/1999/xhtml\">
  <head>\n    <title>Create new author</title>
    <link href=\"/stylesheets/style.css?1149578791\" media=\"screen\"
rel=\"Stylesheet\" type=\"text/css\" />
  </head>
  <body>


  </body>
</html>".
<nil> is not true.

1 tests, 2 assertions, 1 failures, 0 errors
```

We get a failure, because there wasn't a level 1 heading on the returned page. This was to be expected, since we haven't implemented the page yet.

■ **Note** Notice that there was only one failure, even though the page doesn't contain a form element either. This is because a test method is aborted upon the first failure.

Creating the Form

Now we need to make the test pass. Open the controller file, `app/controllers/admin/author_controller.rb`. You can see that the file has a method for each of the actions we specified in the `generate controller` command. We're now interested in the `new` action, which outputs the form for creating a new author. All we need in the `new` action is a new `Author` object. We also set the `@page_title` variable (used by the default layout file described in Chapter 1) to something meaningful.

```
class Admin::AuthorController < ApplicationController
  def new
    @author = Author.new
    @page_title = 'Create new author'
  end

  def create
  end

  def edit
  end

  def update
  end

  def destroy
  end

  def show
  end

  def index
  end
end
```

Next, we need to create the view for the `new` action. Open `app/views/admin/author/new.rhtml` and add the following template code to it:

```
<%= form_tag :action => 'create' %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Create' %>
<%= end_form_tag %>

<%= link_to 'Back', :action => 'index' %>
```

■ **Tip** Using `render :partial => 'partial_name'` is a great way to avoid code duplication in Rails templates. The method call finds a template file `_partial_name.rhtml` and renders it as a part of the surrounding template. This way we can, for example, use the same partial form template for both creating a new author and editing an existing author. If used together with a `:collection` option, the call will render the template once for each element of the container passed with the option. You will see an example of this in the "Listing Authors" section later in this chapter, where we output a similar table row for each author we have in the system. More information about rendering partials in Rails can be found in the Rails API docs at `http://api.rubyonrails.org/classes/ActionController/Base.html#M000206`.

The template first creates a `<form>` tag with the `form_tag` helper, pointing it to the `create` action. Then it renders the actual form, which we will implement in a minute using a partial template. Next, our template outputs a submit button for the form and closes it. Finally, we show a link to the index page listing all the authors (although we haven't implemented that page yet). Next, we need to create the partial template for the actual form. Create a new file `app/views/admin/author/_form.rhtml` and add the following form to it:

```
<%= error_messages_for 'author' %>

<p><label for="author_first_name">First name</label><br/>
<%= text_field 'author', 'first_name'  %></p>

<p><label for="author_last_name">Last name</label><br/>
<%= text_field 'author', 'last_name'  %></p>
```

In the form, we use two useful Rails helper methods:

- `error_messages_for` outputs the validation error notification if there are problems with the form input.
- `text_field` is a form helper that creates a text field element for a given object attribute—in this case, author's first and last names. The real value of `text_field` can be seen when a user inputs something invalid to the form. Rails will automatically mark the field with CSS class `fieldWithErrors`, so you can use CSS to make the field stand out from other fields.

If we run our functional test again, we can see it pass. Opening `http://localhost:3000/admin/author/new` in a browser, we can see that we have implemented the first part of adding a new author, as shown in Figure 2-1.

Figure 2-1. *Form for creating a new author*

Creating the Author

The next thing to do is to actually create the author. For this, we use the `create` action, where the form on the `new` page is pointing. What we want from the create functionality is that when we post a form to the `create` action, a new author is created according to the parameters we specify. Let's extend our tests to test for this, too, by adding a new test case method.

```
def test_create
  assert_equal 0, Author.find(:all).size
  post :create, :author => {:first_name => 'Joel',
                            :last_name => 'Spolsky'}
  assert_response :redirect
  assert_redirected_to :action => 'index'
  assert_equal 1, Author.find(:all).size
  assert_equal 'Author Joel Spolsky was successfully created.', flash[:notice]
end
```

FLASH

Flash (http://api.rubyonrails.org/classes/ActionController/Flash/FlashHash.html) is a clever mechanism in Rails (having nothing to do with Adobe/Macromedia Flash) for sending user notices and error messages from page to page. When we, for example, create a new author, it is a good idea to show the user a notice that the author was indeed created.

Flash is a hash-like structure stored on the server for the time of *one sequential action*. It is available in both controllers and views. So when we set `flash[:notice]` in the `create` action and then redirect to the `index` action, the index page has access to the notice. If the user then clicks a link and goes on to another page, the notice has disappeared from the flash and the notice is not shown anymore.

If you sometimes want to show a flash message in the current action so that it will not be available in the next action (for example, when a form input has been invalid and you want to rerender the form instead of redirecting to the next action), you should use `flash.now` instead of `flash`. It will cause the message to be cleaned up immediately after the current action has been processed.

Here, we first test that there are no authors in the test database. Then we try to create a new author by simulating the HTTP `POST` method sending a form to the `create` action. We expect that Rails responds to the `POST` request by redirecting us to the index page. We also want to make sure that a new `Author` object is created and that Rails sets the `flash[:notice]` variable correctly.

However, the test code is not really beautiful. Instead of checking the exact amount of authors before and after running the `create` action, we only want to know that the amount was incremented by one. Fortunately, the chaps at projectionist (`http://project.ioni.st/`) have created a helper assertion method for testing just that (`http://project.ioni.st/post/218#post-218`). Open `test/test_helper.rb` in a text editor and add the following code to it:

```
def assert_difference(object, method, difference=1)
  initial_value = object.send(method)
  yield
  assert_equal initial_value + difference,
    object.send(method)
end

def assert_no_difference(object, method, &block)
  assert_difference object, method, 0, &block
end
```

■ **Tip** `test/test_helper.rb` is a file where you can define helper methods that are available to all tests. A helper method can be a custom assertion like above, or any other method that makes writing actual tests easier and cleaner.

`assert_difference` takes as its parameters an object, a method of that object, and a code block. It first stores the initial value returned by `object.method`, then runs the code block, and after that fetches the return value of `object.method` again. If the difference between those two method calls is not exactly the value of the `difference` parameter (`1` by default), the test will fail. `assert_no_difference` is a convenience method that calls `assert_difference` with the `difference` value of zero.

Now we can change the code in `test_create` to the following:

```
def test_create
  get :new
  assert_template 'admin/author/new'
  assert_difference(Author, :count) do
    post :create, :author => {:first_name => 'Joel',
                              :last_name => 'Spolsky'}
    assert_response :redirect
    assert_redirected_to :action => 'index'
  end
  assert_equal 'Author Joel Spolsky was successfully created.', flash[:notice]
end
```

Now, instead of explicitly checking the number of authors before and after the request, we enclose it inside an `assert_difference` call. And because the default value of the `difference` parameter is `1`, `assert_difference` passes if, and only if, the code inside its code block increments the count of authors by one. It doesn't matter what the value was before, so we're not relying on the `authors` table being empty at the start anymore.

Running the test obviously fails, so let's move on to implement the author creation. We have an empty `create` action in `author_controller.rb`, so we can go ahead and fill it in.

```
def create
  @author = Author.new(params[:author])
  if @author.save
    flash[:notice] = "Author #{@author.name} was successfully created."
    redirect_to :action => 'index'
  else
    @page_title = 'Create new author'
    render :action => 'new'
  end
end
```

The `create` action is a typical example of how actions that modify data work in Rails. First, we create a new `Author` object from the request parameters sent from the form on the `new` page. Then we try to save the object. If the object was valid and could thus be saved, we create a flash

message to be shown to the user and redirect to a page that lists all the authors. If `@author` couldn't be saved—probably because it didn't pass the validations (we'll talk more about validations in the next section)—we render the `new.rhtml` template with the form again, so that the user can fill in the required fields she forgot to fill in the first time.

Running the test again, we see that it passes. But George has also advised us that every author needs to have both a first and a last name. We thus want to test that no new author is created if either of those form fields is left empty. We create another test method for this in `test/functional/admin/author_controller_test.rb`:

```
def test_failing_create
  assert_no_difference(Author, :count) do
    post :create, :author => {:first_name => 'Joel'}
    assert_response :success
    assert_template 'admin/author/new'
    assert_tag :tag => 'div', :attributes => {:class => 'fieldWithErrors'}
  end
end
```

Here, we do the same thing as in the previous test, except that this time, we leave the last name out of the form post. Now, instead of a redirect, we want Rails to show us the form template again, with the fields with errors marked accordingly. We also use `assert_no_difference` to make sure that the new author is not created.

When we run the test, we can see it failing. Instead of rendering the form again on an invalid form input, we are still redirected to the index page. We need a way to make the form validate only if both the first and the last name are present. In Rails, validations are done on the ActiveRecord object level.

Validating Data

ActiveRecord validations are a way to enforce restrictions upon editing the business model objects of your application. For example, you might want to make sure that an e-mail address is valid or that the balance of an account can never be less than zero.

In our case, we just want to make sure that every time someone tries to add or update an `Author` object, that object has both the first and the last name specified. This can be accomplished with the `validates_presence_of` method, so let's open `app/models/author.rb` and add a line of code there to make the validations work:

```
class Author < ActiveRecord::Base
  validates_presence_of :first_name, :last_name

  def name
    "#{first_name} #{last_name}"
  end
end
```

We will meet a lot more validations during the course of this book, but for now, let's just settle with this and try to run our tests again.

```
$ ruby test/functional/admin/author_controller_test.rb
```

```
Loaded suite test/functional/admin/author_controller_test
Started
..
Finished in 0.171436 seconds.

2 tests, 12 assertions, 0 failures, 0 errors
```

The results look good. We can now point our browser to `http://localhost:3000/admin/author/new` and try to create a new author. Leaving the last name blank, we're greeted with the response shown in Figure 2-2 when the form is submitted.

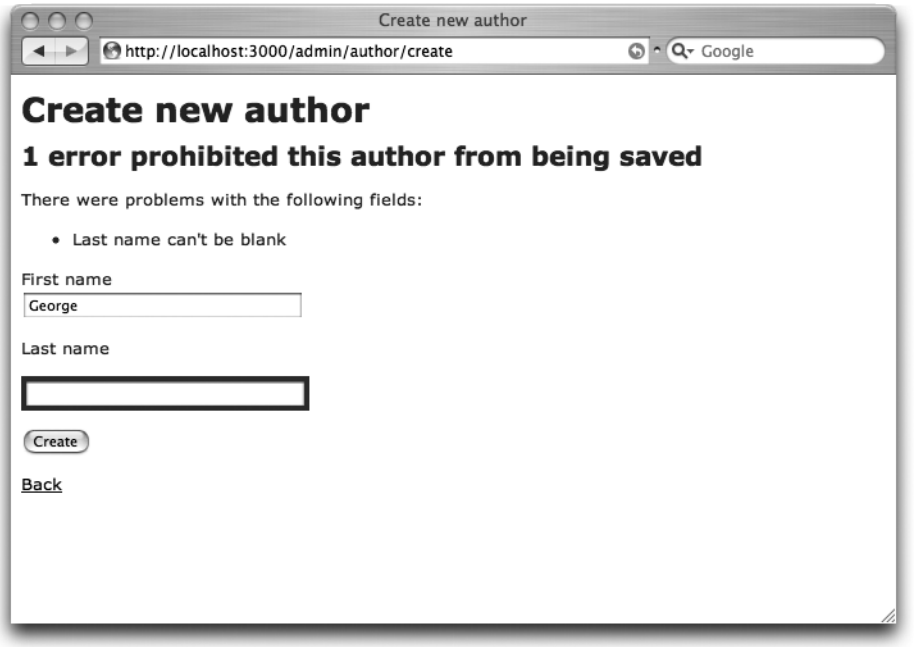


Figure 2-2. *Failing creation of a new author*

Seems that our validations are working, and as the tests agree, we move on to implement editing of existing authors.

Listing Authors

Now that we are able to create authors, it would be nice to be able to view and list the authors in the system, too. We will next implement the author list page, which we'll make the `index` action of our controller.

At its simplest, we want our author list to show a table with rows for all authors and a header row. Let's create a test case for it in `author_controller_test.rb`.

```
def test_index
  get :index
  assert_response :success
  assert_tag  :tag => 'table',
              :children => { :count => Author.count + 1,
                             :only => {:tag => 'tr'} }
  Author.find(:all).each do |a|
    assert_tag  :tag => 'td',
                :content => a.first_name
    assert_tag  :tag => 'td',
                :content => a.last_name
  end
end
```

In the test, we first check that the `index` action returns a successful HTTP response. Then we check that there is a table with rows for each author in the database and a header row on the resulting page. We do this by using the `:children` and `:count` options of `assert_tag`. We also check that there are table cells holding the names of all authors in the database.

Implementing the `index` action is straightforward. In the controller, we just fetch all the authors from the database and set the page title:

```
def index
  @authors = Author.find(:all)
  @page_title = 'Listing authors'
end
```

In the view file, `app/views/admin/author/index.rhtml`, we then display the table and, at the bottom of the page, a link to add a new author:

```
<table>
  <tr>
    <th>Name</th>
    <th>Edit</th>
    <th>Delete</th>
  </tr>

  <%= render :partial => 'author', :collection => @authors %>
</table>

<p><%= link_to 'Add a new author', :action => 'new' %></p>
```

This time, we use the `:collection` option in the `render` call to render the `_author.rhtml` partial template once for each author in `@authors`. Let's create the partial template (`app/views/admin/author/_author.rhtml`) and make it show a table row for each author.

```
<tr>
  <td><%= link_to author.name, :action => 'show', :id => author %></td>
  <td><%= link_to 'Edit', :action => 'edit', :id => author %></td>
  <td>
    <%= button_to 'Delete', {:action => 'destroy', :id => author},
        :confirm =>
          "Are you sure you want to delete author #{author.name}?" %>
  </td>
</tr>
```

On each row, we show the name of the author linking to an individual show page (which we'll implement soon) and a link to edit the author details. In the last cell, we use the `button_to` helper to show a form button for deleting the author record (we will implement the action for deleting authors later in this chapter). We pass the method call a `:confirm` option, which causes the browser to ask the user for a confirmation with JavaScript when the Delete button is clicked.

We can now run the test file again and see that everything seems to be in order. Now we will move on to the next user story, viewing the details of an individual author.

Viewing an Author

As you might have noticed, creating the `Author` model also created a file called `authors.yml` in `test/fixtures`. This is called a fixture file. *Fixtures* are mock data that can be used to populate the database with consistent data before each test method. Since the test database is purged before every test method, you know that all the data that exists in the database at that point came from the fixture files.

It would be handy to have a few authors in the database for testing our view functionality, so we go ahead and create a couple of author fixtures in `authors.yml`:

```
joel_spolsky:
  id: 1
  first_name: Joel
  last_name: Spolsky
jeremy_keith:
  id: 2
  first_name: Jeremy
  last_name: Keith
```

Putting the line `fixtures :authors` in the beginning of our functional test class makes Rails load the author fixtures automatically before every test method inside that class:

```
class Admin::AuthorControllerTest < Test::Unit::TestCase
  fixtures :authors
...
end
```

Now we can rest assured that when we start testing viewing an author, we have two items in our `authors` table.

We'll keep the show author page very simple. We just want to make sure that we're fed the right template and that the author is the one we're expecting. Add the following test case to the bottom of `author_controller_test.rb`:

```
def test_show
  get :show, :id => 1
  assert_template 'admin/author/show'
  assert_equal 'Joel', assigns(:author).first_name
  assert_equal 'Spolsky', assigns(:author).last_name
end
```

Here, we simply request the show page for one of our fixture authors and check that we get the correct template. Then we use the `assigns` helper to check that the author instance variable assigned in the action is the one it should be. `assigns` is a test helper method that can be used to access all the instance variables set in the last requested action. Here, we expect that the `show` action assigns a variable `@author` and that the variable responds to the methods `first_name` and `last_name`, returning "Joel" and "Spolsky," respectively.

The controller code for the `show` action is simple. We fetch the author from the database and set the page title to the author's name.

```
def show
  @author = Author.find(params[:id])
  @page_title = @author.name
end
```

Now let's open the view file, `app/views/admin/author/show.rhtml`, and add the template code:

```
<dl>
  <dt>First Name</dt>
  <dd><%= @author.first_name %></dd>
  <dt>Last Name</dt>
  <dd><%= @author.last_name %></dd>
</dl>

<p><%= link_to 'Edit', :action => 'edit', :id => @author %> |
<%= link_to 'Back', :action => 'index' %></p>
```

We run the tests again, and seeing them passing, open the browser and check that the page looks fine there, too, as shown in Figure 2-3.

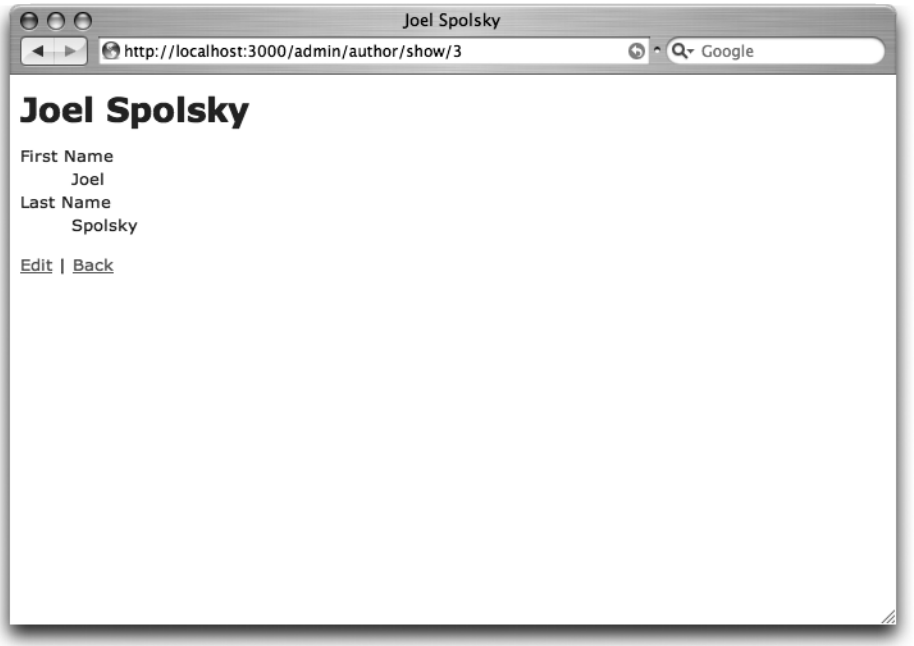


Figure 2-3. *Show author page*

Editing an Author

We start our test case for the Edit Author user story in the same way we did with the test case for the create functionality, by loading the form page.

```
def test_edit
  get :edit, :id => 1
  assert_tag  :tag => 'input',
              :attributes => { :name => 'author[first_name]',
                               :value => 'Joel' }
  assert_tag  :tag => 'input',
              :attributes => { :name => 'author[last_name]',
                               :value => 'Spolsky' }
end
```

We try to load the edit page for author 1 and test that the form has input fields with correct preset values. After running the test and finding it running red, we implement the edit interface. We first set the needed values in the `edit` action in `app/controllers/admin/author_controller.rb`:

```
def edit
  @author = Author.find(params[:id])
  @page_title = 'Edit author'
end
```

We use the normal `ActiveRecord` `find` method to fetch the `Author` object with the `id` that was passed with the URL. Then we populate the page title, just as we did with the `new` page. The view file `app/views/admin/author/edit.rhtml` looks pretty much the same as `new.rhtml`, with the exception that the form action is `update` instead of `create` this time. Note that we use the same partial template to output the actual form fields as on the `new` page.

```
<%= start_form_tag :action => 'update', :id => @author %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= link_to 'Show', :action => 'show', :id => @author %> |
<%= link_to 'Back', :action => 'index' %>
```

We run the tests again, and everything should pass. Next, we need to implement the action that receives the edit form post, `update`. We extend our tests a bit:

```
def test_update
  post :update, :id => 1, :author => { :first_name => 'Joseph',
                                       :last_name => 'Spolsky' }
  assert_response :redirect
  assert_redirected_to :action => 'show', :id => 1
  assert_equal 'Joseph', Author.find(1).first_name
end
```

In the update, we post the form to the `update` action, and after that, check that we are redirected and, more important, that the first name of the author is really changed. The test doesn't yet pass, of course, so let's open the `author_controller.rb` file again and implement the `update` action.

```
def update
  @author = Author.find(params[:id])
  if @author.update_attributes(params[:author])
    flash[:notice] = 'Author was successfully updated.'
    redirect_to :action => 'show', :id => @author
  else
    @page_title = 'Edit author'
    render :action => 'edit'
  end
end
```

You can see that the action is more or less similar to the `create` action, with the distinction that, this time, we use the `update_attributes` method to update the `@author` object. It updates the attributes of the object with the values it gets from the form parameters, and after that, it calls `save` implicitly. Just like `save`, it returns `false` if saving the object didn't succeed.

We can now run the test file again, and see that everything passes just fine.

```
$ ruby test/functional/admin/author_controller_test.rb
```

```
Loaded suite test/functional/admin/author_controller_test
Started
...
Finished in 0.519315 seconds.

3 tests, 28 assertions, 0 failures, 0 errors
```

Deleting an Author

For the Delete Author user story, we simply want to test that posting a form to the `destroy` action correctly deletes the author from the database and then redirects us to the `index` page. Let's open `author_controller_test.rb` again and add another test case to it.

```
def test_destroy
  assert_difference(Author, :count, -1) do
    post :destroy, :id => 1
    assert_response :redirect
    assert_redirected_to :action => 'index'
  end
end
```

The test code is fairly simple, and the only thing new is that this time we pass `-1` as the `difference` argument to `assert_difference`, in order to make sure the number of authors decreases by one as the result of the `destroy` action.

Implementing the `destroy` action is straightforward. We don't need a view for it, since the action just deletes the author and redirects back to the author list.

```
def destroy
  @author = Author.find(params[:id])
  flash[:notice] = "Successfully deleted author #{@author.name}"
  @author.destroy
  redirect_to :action => 'index'
end
```

That's all the code it takes to delete an author. We again fetch the right author from the database using the `find` method, and then use the ActiveRecord `destroy` method to delete the author from the database. In between, we populate the flash notice so that the `index` page where we redirect the user will show a notification to the user about the successful deletion.

We can now run the whole test enchilada.

```
$ ruby test/functional/admin/author_controller_test.rb
```

```
Loaded suite test/functional/admin/author_controller_test
Started
......
Finished in 0.176094 seconds.

6 tests, 31 assertions, 0 failures, 0 errors
```

The result is 31 assertions, no failures, no errors. It works! It works!!! We exchange a couple of high-fives before we start the WEBrick server again, and point the browser to `http://localhost:3000/admin/author/`. The result is shown in Figure 2-4.

Figure 2-4. *Author list page*

Adjusting the Flash Notifications

Browsing around the interface—creating, editing, and deleting authors—we can see that everything works. However, we find one shortcoming. The flash notifications we used to display messages to the user are not shown. Indeed, while we did assign the messages, we are never displaying them in the user interface.

Again, we create a failing test assertion demonstrating this flaw. We could put the new test in all the test methods that should be showing a notice using `flash`, but we settle for doing it in the author deletion test.

```
def test_destroy
  assert_difference(Author, :count, -1) do
    post :destroy, :id => 1
    assert_response :redirect
    assert_redirected_to :action => 'index'
    follow_redirect
    assert_tag :tag => 'div', :attributes => {:id => 'notice'},
          :content => 'Successfully deleted author Joel Spolsky'
  end
end
```

follow_redirect is a Rails test helper method that causes the test case to follow a redirect_to call in a controller. In our case, the destroy action redirected the user to the index action in the end, so follow_redirect causes the index action to be run. After that we can check that there is a div tag with the correct flash message on the index page.

Since flash messages can be shown to the user on many different pages, a natural place for displaying the notice is in the layout template used by all actions, app/views/layouts/application.rhtml. We add a bit of code there that will show the message stored in flash[:notice] if it is assigned.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title><%= @page_title || 'Emporium' %></title>
    <%= stylesheet_link_tag "style" %>
  </head>
  <body>
    <% if flash[:notice] %>
      <div id="notice">
        <%= flash[:notice] %>
      </div>
    <% end %>
    <%= "<h1>#{@page_title}</h1>" if @page_title %>
    <%= yield %>
  </body>
</html>
```

We also make a small change to the CSS file created in Chapter 1, public/stylesheets/style.css, adding a bit of color to make the notice stand out more on the page:

```
… a lot of CSS lines omitted …
#notice {
  padding: 5px;
  background-color: #96FF88;
}
```

Running the test again indicates that the flash system works, and so does our empirical research done in the browser, as shown in Figure 2-5.

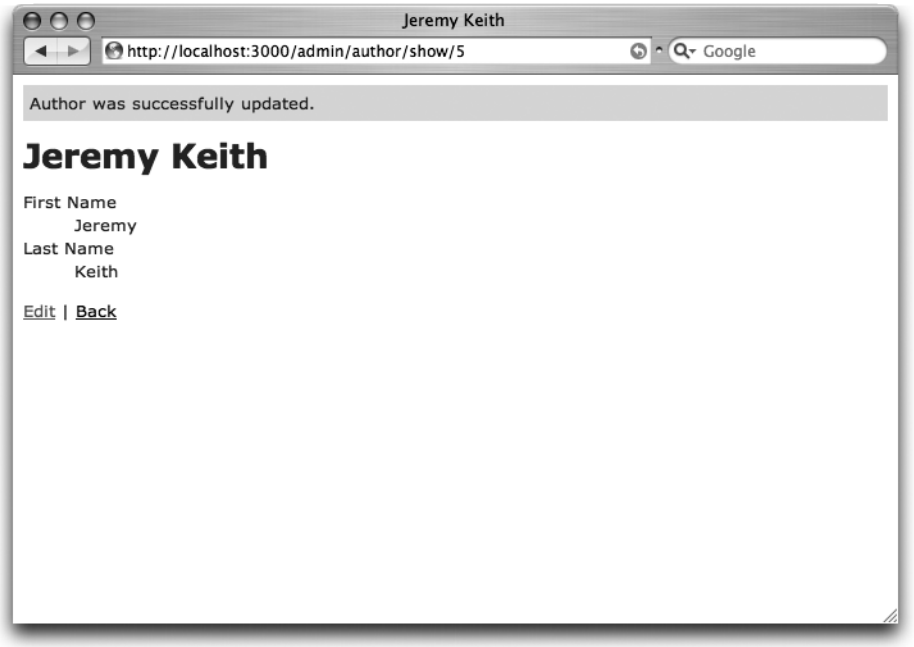


Figure 2-5. *Working flash message*

Checking off the last thing as done from our sprint backlog, we call George to do the acceptance testing. George can't believe we're finished already. "Holy moly, I didn't even have the time to finish my lunch yet," he says, mustard dripping from his beard. Time spent for this sprint: three hours. Now that's productivity!

Summary

In this chapter, we first introduced the approach of TDD and the testing methods Ruby on Rails supports. Then we continued with putting the TDD in action in Rails, creating a controller for handling adding, updating, deleting, and viewing authors. During the course of the chapter, we also briefly introduced ActiveRecord object-relational mapping, Rails migrations, and ActiveRecord validations. These techniques give you a solid foundation for managing the business logic of your Rails applications.

In the next chapter, you will learn that we could have done the same tasks we did in this chapter with a single command, by using the Rails scaffolding. But that's okay. Doing it manually first, we could better introduce some useful Rails features and, most of all, run a code sprint in a true TDD fashion.

CHAPTER 3

Book Inventory Management

In this chapter, we will quickly implement a complete book inventory management system using the built-in scaffolding feature of Ruby on Rails. While implementing the system, we will write Ruby on Rails integration tests that exercise the whole book inventory management system from end to end. As we work through this sprint, we will show you how to map database relationships with ActiveRecord, including one-to-many, many-to-one, and many-to-many relationships. Plus, you will learn how to implement file upload functionality with Rails. We'll also introduce you to the Textile markup language, which can be used to author web content.

Getting the Requirements

To figure out the requirements for this sprint, we ask George what tasks he should be able to perform with the book management system. He tells us that he receives information from publishers about when new books are published and when old ones go out of print. He asks us to build a system that will allow him to update the Emporium book inventory accordingly.

George tells us that he is a big fan of Amazon, even if the giant is eating away at his profits. He likes, for example, the way Amazon is able to recommend books that are similar to what the customer has browsed and bought before, and the small details like the book cover shown next to the details of the book.

We tell George that he at least has to be able to find books in the system, and to view and edit the book details. George confirms that he indeed needs those features. We take a short break, to let George have a coffee. This far into the discussion we have identified the following user stories:

- *Add book*: George, the administrator of Emporium, must be able to add new books to the inventory.
- *Upload book cover*: The administrator must be able to upload an image of the book cover. This image will be shown to customers.
- *List books*: The administrator must be able to list all of the books that are currently available in the inventory.
- *View book*: The administrator must be able to view the details of a book.
- *Edit book*: The administrator must be able to edit the details of a book.
- *Delete book*: The administrator must be able to delete books from the inventory.

We aren't sure how we should implement the book recommendation feature George wanted, so we decide to postpone that for a later sprint (covered in Chapter 7).

When George comes back, he tells us that he must also be able to keep track of the book publishers. After a brief brainstorming session, we come up with a list of user stories related to publisher management:

- *Add publisher*: The administrator must be able to add publishers to the system.
- *List publishers*: The administrator must be able to view a list of all publishers in the system.
- *View publisher*: The administrator must be able to view the details of a publisher.
- *Edit publisher*: The administrator must be able to edit the details of a publisher.
- *Delete publisher*: The administrator must be able to remove publishers from the system.

We will also implement these in this sprint, as we are confident that we can finish them within the schedule.

Using Scaffolding

In this chapter, we'll show you how to use a built-in Rails feature called *scaffolding* to jump-start the implementation of the publisher administration and book administration user interfaces.

You can use scaffolding to generate a complete CRUD implementation for objects stored in a database. Scaffolding can generate code for all three MVC layers: the model, view, and controller. Scaffolding comes in two flavors:

- *Static scaffolding* creates the code physically on disk. Static scaffolding is suited for generating boilerplate code, which you can modify later to fit your requirements.
- *Dynamic scaffolding* creates the code in memory only. It does the same as static scaffolding but at runtime, and no files are generated. This is suited only for simple functionality, such as an interface used only by administrators for editing a list of options shown to users.

The scaffolding script accepts a set of parameters that tells it what to generate. You can specify the model, view, controller, and actions as parameters to the scaffold script:

```
script/generate scaffold ModelName ControllerName action1 action2
```

When invoked, the scaffolding script connects to the database and inspects the table structure for the specified model. It then creates the controller, actions, and views necessary for creating, viewing, editing, and deleting the model you specified.

Dynamic scaffolding is done by adding a call to the `scaffold` method to a controller:

```
class JebusController < ActionController::Base
  scaffold :jebus
end
```

Scaffolding can't generate code that fits your requirements perfectly, so we'll also show you how to modify and extend the generated code.

■ **Tip** Scaffolding provides examples of Rails best practices and coding conventions. It's a good idea to have a closer look at the code that is generated with scaffolding, even if you don't plan on using scaffolding.

Implementing the Publisher Administration Interface

We will start by implementing the administrator interface for maintaining the list of publishers. We need a table for storing publishers, so the first thing we need to do is to update the database schema by adding the `publishers` table to the database schema.

Updating the Schema with the Publishers Table

As in the previous chapter, we will use ActiveRecord migrations to make the necessary modifications to the database schema. We could also use plain SQL, but migrations have the added benefit of being database-agnostic and allowing you to roll back changes.

First, create the `create_publishers` migration file, which you will use for adding the `publishers` table to the database schema:

```
$ script/generate migration create_publishers
```

```
      exists  db/migrate
      create  db/migrate/002_create_publishers.rb
```

Open `db/migrate/002_create_publishers.rb` in your editor and change it as follows:

```
class CreatePublishers < ActiveRecord::Migration
  def self.up
    create_table :publishers do |table|
      table.column :name, :string, :limit => 255, :null => false, :unique => true
    end
  end

  def self.down
    drop_table :publishers
  end
end
```

The migration will create a table named `publishers` when run, as the following sample output shows:

```
$ rake db:migrate
```

```
(in /home/george/projects/emporium)
== CreatePublishers: migrating ================================================
-- create_table(:publishers)
   -> 0.2030s
== CreatePublishers: migrated (0.2030s) =======================================
```

The new table has two columns: `id` and `name`. Note that the `id` column is automatically added by ActiveRecord migrations, so we need to add only the `name` column to the migration script. We limit the `name` column's length to a maximum length of 255 characters. We also specify that we don't accept null values in the `name` field and that the name must be unique.

Following good practices, we undo all changes in the `down` method by telling ActiveRecord to delete the `publishers` table.

Generating Publisher Code with the Scaffolding Script

With the database table in place, you can use the scaffolding script to create an almost complete CRUD implementation for the publisher administration, unlike in Chapter 2 where all code was created by hand. Execute the scaffolding script as follows:

```
$ script/generate scaffold Publisher 'admin/publisher'
```

```
      exists  app/controllers/admin
      exists  app/helpers/admin
      create  app/views/admin/publisher
      exists  test/functional/admin
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/publisher.rb
      create    test/unit/publisher_test.rb
      create    test/fixtures/publishers.yml
      create  app/views/admin/publisher/_form.rhtml
      create  app/views/admin/publisher/list.rhtml
      create  app/views/admin/publisher/show.rhtml
      create  app/views/admin/publisher/new.rhtml
      create  app/views/admin/publisher/edit.rhtml
      create  app/controllers/admin/publisher_controller.rb
      create  test/functional/admin/publisher_controller_test.rb
      create  app/helpers/admin/publisher_helper.rb
      create  app/views/layouts/publisher.rhtml
   identical  public/stylesheets/scaffold.css
```

The scaffolding script creates the model, controller, and views required for doing CRUD operations on the `publishers` table. Furthermore, the scaffolding creates an empty unit test for the model, along with a fixture file and a functional test.

Next, we demonstrate the wonders of scaffolding to George. He says: "Damned consultants! I'm gonna go blind if I have to look at that page for more than ten seconds! Is this all you can do?" We understand his point, and show him the new site design we have quietly been working on, to which he responds, "It's not going to win any design awards. But, it will do until I can find the money to hire a professional web designer."

■ **Note** To get the new layout, download the source code for this book from the Apress website (`www.apress.com`), and copy the layout file (`application.rhtml`) and style sheet file (`style.css`) to your project directory.

Next, start WEBrick, if it isn't running already, by executing `script/server` in the root directory of the application. Open `http://localhost:3000/admin/publisher` in your browser. You should see the user interface for listing publishers. We'll do a small test just to verify that the generated code works. Click the New publisher link, enter the name Apress in the Name field, and then click Create. You should now see a success message telling you that the publisher was created successfully. You should also see a new row in the list of publishers, as shown in Figure 3-1.

Figure 3-1. *The publisher list page after adding a publisher*

Scaffolding creates some files that we don't need, including a style sheet and a layout file. Delete the `public/stylesheets/scaffold.css` file, as we already have a style sheet for Emporium. Also delete `app/views/layouts/publisher.rhtml`, as we want to use the same layout for all controllers.

Now is a good time to start writing some tests. But wait, didn't the scaffolding script just create a functional test for us? Let's execute the test with `rake test:functionals`. There are no errors and all tests pass. You could assume that you do not have to write any tests, but that assumption is wrong.

■ **Note** Because we are using scaffolding, we won't be following TDD very strictly in this chapter.

Completing the Add Publisher User Story

The scaffolding script has created a functional test in `test/functional/admin/publisher_controller_test.rb`. On closer inspection, we can see that it requires some modifications for it to be helpful in our efforts at producing bug-free code. For example, the `test_create` method doesn't specify a name for the publisher it creates. This should have made the test fail when we ran the test, but it didn't. We will fix that soon, but the first thing we will do is add validations. You never know what kind of data a user will try to enter into your application. As explained in the previous chapter, validations help ensure that only valid data is inserted into the database.

Adding Validations to the Model

Adding validations will make the test fail and show you where the parameters should be specified. So let's begin by adding a validation for the `name` field in the `Publisher` model. Open `app/models/publisher.rb` and modify it to look as follows:

```
class Publisher < ActiveRecord::Base
  validates_length_of :name, :in => 2..255
  validates_uniqueness_of :name
end
```

As you might remember, we specified the maximum length of the `name` field in the `publishers` table to be 255 characters. We add a validation to the model to verify this constraint and specify that the minimum length of the `name` field to be 2 characters. We also add a validation that checks that the publisher name is unique.

Modifying the Generated Fixture Data

The fixture data generated by scaffolding is not very descriptive of our project. We can do better. Open `test/fixtures/publishers.yml` in your editor and remove everything from the file. Then add the following:

```
apress:
  id: 1
  name: Apress
```

Again, execute the functional tests with `rake test:functionals`. You should see the tests fail with the following error message:

```
Expected response to be a <:redirect>, but was <200>
```

The test fails as expected because the `test_create` method doesn't provide any parameters to the `post` method that is supposed to create the publisher. To fix this, we'll change the functional test.

Modifying the Generated Functional Test

Open `test/functional/admin/publisher_controller_test.rb` in your editor and change the `test_create` method as follows:

```
  def test_create
    num_publishers = Publisher.count

    post :create, :publisher => {:name => 'The Monopoly Publishing Company'}

    assert_response :redirect
    assert_redirected_to :action => 'list'

    assert_equal num_publishers + 1, Publisher.count
  end
```

The only change is that we pass a hash to the `post` method, instead of no data at all. The hash contains the name for the publisher the test should create.

Run the tests again, and you should see the functional test pass. You should also do a quick test in the browser to verify that the Add Publisher user story functionality works. Open `http://localhost:3000/admin/publisher/new` in your browser. Test the validations you just added by clicking the Create button, without specifying a name. You should see the error message shown in Figure 3-2. This error message is automatically generated by Rails, and explains exactly what you should do in order to fix the error.

Figure 3-2. *Testing the Add Publisher user story*

We further examine the functional tests and see that those for the List Publishers and Delete Publisher user stories are satisfactory. However, the test for the View Publisher and Edit Publisher user stories require some work.

Completing the View Publisher User Story

We are satisfied with the functional test that scaffolding created for the View Publisher user story, except for one detail. It doesn't verify that the view really is showing the details of the publisher. This will be easy to fix, but first have a look at the view.

Modifying the View

Open app/views/admin/publisher/show.rhtml in your editor, and you can see that it is not using the same style as the View Author user story we created in the previous chapter. Change the view as follows, so that it uses the same style:

```
<dl>
  <dt>Name</dt>
  <dd><%= @publisher.name %></dd>
</dl>

<%= link_to 'Edit', :action => 'edit', :id => @publisher %> |
<%= link_to 'Back', :action => 'list' %>
```

Modifying the Generated Action

Recall that we modified the layout file (application.rhtml) in the previous chapter to display the page title, if it is made available to the view. Currently, this is not the case for the show publisher page, as can be verified by viewing the details of a publisher. Fix this by opening app/controllers/admin/publisher_controller.rb and changing the show method as follows:

```
def show
  @publisher = Publisher.find(params[:id])
  @page_title = @publisher.name
end
```

This allows the view to access the instance variable @page_title and print out the value.

Modifying the Generated Functional Test

Next, modify the generated test so that it verifies that the page is rendered correctly. Open test/functional/admin/publisher_controller_test.rb and change it as follows:

```
def test_show
  get :show, :id => 1

  assert_response :success
  assert_template 'show'

  assert_not_nil assigns(:publisher)
  assert assigns(:publisher).valid?

  assert_tag "h1", :content => Publisher.find(1).name
end
```

Note that we added an `assert_tag` assertion to the end of the test. This assertion is used to verify that the main heading on the page is showing the publisher's name.

Run the functional tests again, and you should see all tests pass without errors. Access `http://localhost:3000/admin/publisher/list` and click the Show link next to the publisher you just created. You should see a page that looks like Figure 3-3.

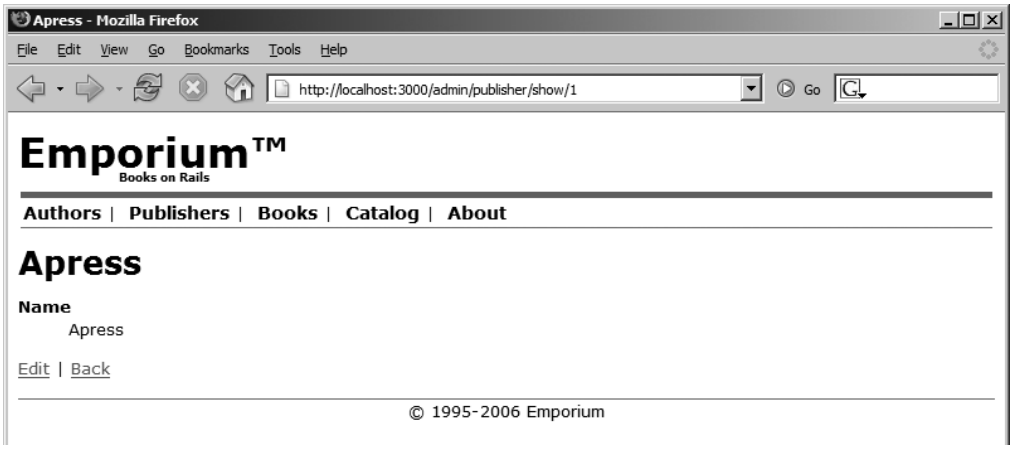


Figure 3-3. *Testing the View Publisher user story*

Completing the Edit Publisher User Story

The scaffold implementation of the Edit Publisher user story's test suffers from the same problem as the Add Publisher implementation. The test doesn't provide any parameters to the action other than the `id` of the publisher. This means that no data is updated when the test is run, but in this case, there is no error. This is okay, because Rails uses `update_attributes` to update only attributes that are included as request parameters.

We do want to verify that the editing is successful, so open `test/functional/admin/publisher_controller_test.rb` and change the `test_update` method as shown in the following code snippet:

```
def test_update
  post :update, :id => 1, :publisher => { :name => 'Apress.com' }
  assert_response :redirect
  assert_redirected_to :action => 'show', :id => 1
  assert_equal 'Apress.com', Publisher.find(1).name
end
```

Note that we have added a new parameter to the `post` method call. This will update the name of the publisher to `Apress.com` in the database. At the end of the test, we verify that this really is the case with `assert_equal`. Execute the functional tests again, with `rake test:functionals`. Access `http://localhost:3000/admin/publisher/list` and click the Edit link next to the publisher you created. You should see a page that looks like Figure 3-4.

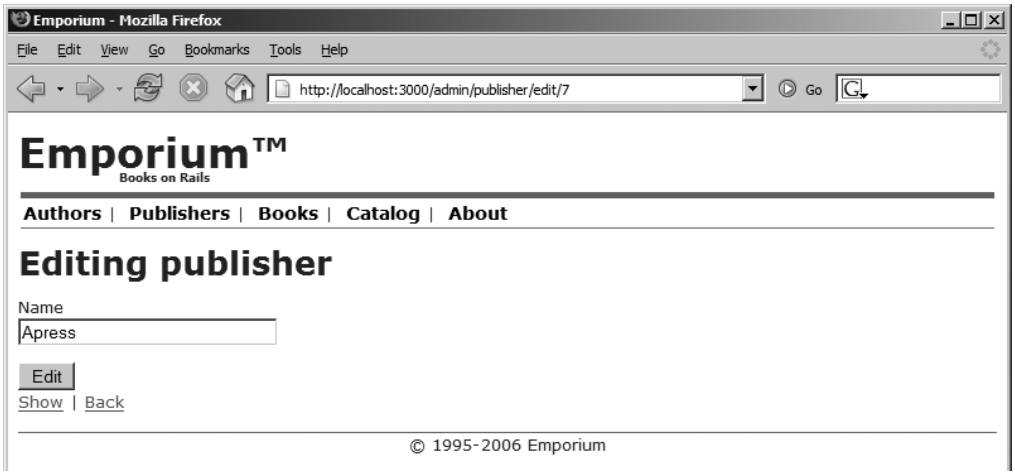


Figure 3-4. *Testing the Edit Publisher user story*

You now have a functioning system for maintaining the publishers. We call George over and show him the user interface. He does a quick acceptance test and tells us that he has no complaints, so let's continue implementing the user stories for book management.

Implementing the Book Administration Interface

Now we implement the administrator interface for managing books. For this, we need to first create a table for storing books in the database, and then create the ActiveRecord model for books. Although we found some issues with scaffolding while implementing the publisher administration functionality, it saved us some time and George was happy, so we'll continue using scaffolding to implement the book management administration functionality.

Updating the Schema with the Books Table

The first thing we need to do is to create a table for storing books in the database. Following the Rails naming conventions, we name the table `books`, and as usual, we'll update the database schema using ActiveRecord migrations.

As a recap, the Emporium database schema already has the `authors` table we created in the previous chapter and the `publishers` table we created earlier in this chapter. Now, we'll add the `books` table, as well as the `books_authors` table, which will be used to link `authors` to `publishers`.

George is still with us, at least physically. He asks us why we have to use so many tables. "You consultants, you always want to build these fancy systems. I could do this with just one Excel sheet!" We don't know if he's kidding or not, but we'll try to get George to understand when we show him a picture and explain how mapping works to link the tables and get the data we want. We'll do that after we add the new tables and create the `Book` model.

To start, create the migration file for adding the `books` table to the database schema:

```
$ script/generate migration create_books_and_authors_books
```

```
      exists  db/migrate
      create  db/migrate/003_create_books_and_authors_books.rb
```

Open `db/migrate/003_create_books_and_authors_books.rb` in your editor and add the code in Listing 3-1 to it.

Listing 3-1. *ActiveRecord Migration for the books and authors_books Tables and Foreign Keys*

```
class CreateBooksAndAuthorsBooks < ActiveRecord::Migration
  def self.up
    create_table :books do |table|
      table.column :title, :string, :limit => 255, :null => false
      table.column :publisher_id, :integer, :null => false
      table.column :published_at, :datetime
      table.column :isbn, :string, :limit => 13, :unique => true
      table.column :blurb, :text
      table.column :page_count, :integer
      table.column :price, :float
      table.column :created_at, :timestamp
      table.column :updated_at, :timestamp
    end

    create_table :authors_books, :id => false do |table|
      table.column :author_id, :integer, :null => false
      table.column :book_id, :integer, :null => false
    end

    say_with_time 'Adding foreign keys' do
      # Add foreign key reference to books_authors table
      execute 'ALTER TABLE authors_books ADD CONSTRAINT fk_bk_authors ➥
FOREIGN KEY ( author_id ) REFERENCES authors( id ) ON DELETE CASCADE'
      execute 'ALTER TABLE authors_books ADD CONSTRAINT fk_bk_books ➥
FOREIGN KEY ( book_id ) REFERENCES books( id ) ON DELETE CASCADE'
```

```
      # Add foreign key reference to publishers table
      execute 'ALTER TABLE books ADD CONSTRAINT fk_books_publishers ➥
FOREIGN KEY ( publisher_id ) REFERENCES publishers( id ) ON DELETE CASCADE'
    end
  end

  def self.down
    drop_table :authors_books
    drop_table :books
  end
end
```

■ **Note** This migration in Listing 3-1 uses MySQL-specific SQL. This means that you would need to change the code in order to run it on other databases.

The migration file creates two new tables, `books` and `authors_books` (`authors_books` in the join table, as explained in the "Many-to-Many Relationship" section later in this chapter). To ensure data integrity, we also add foreign key constraints to the tables. ActiveRecord doesn't support adding foreign keys constraints to tables. You need to add foreign keys using the `ALTER TABLE` SQL command and the ActiveRecord `execute` method, which can execute raw SQL on the database. The `say_with_time` method is used to print out the time it takes to execute the commands that add foreign keys to the database schema. Also note that ISBN numbers must be unique and that this is ensured by setting the `:unique` option to `true`.

■ **Tip** ActiveRecord has a built-in time stamping behavior that is triggered for database columns named `created_at`/`created_on` and `updated_at`/`updated_on`. When ActiveRecord finds one of these columns in a database schema, it will automatically set the creation timestamp when a new object is created and the modification time when the object is updated. ActiveRecord also has other behaviors that are triggered for other column names. For example, the `lock_version` column name enables optimistic locking.

You are now ready to upgrade the Emporium database to its third version. Execute the migration script with the `rake db:migrate` command:

```
$ rake db:migrate
```

```
(in /home/george/projects/emporium)
== CreateBooksAndAuthorsBooks: migrating ======================================
-- create_table(:books)
   -> 0.1410s
-- create_table(:authors_books, {:id=>false})
   -> 0.1400s
-- Adding foreign keys
-- execute("ALTER TABLE authors_books ADD CONSTRAINT fk_bk_authors ➥
FOREIGN KEY ( author_id ) REFERENCES authors( id ) ON
 DELETE CASCADE")
   -> 0.3440s
-- execute("ALTER TABLE authors_books ADD CONSTRAINT fk_bk_books ➥
FOREIGN KEY ( book_id ) REFERENCES books( id ) ON DELETE CASCADE")
   -> 0.3280s
-- execute("ALTER TABLE books ADD CONSTRAINT fk_books_publishers ➥
FOREIGN KEY ( publisher_id ) REFERENCES publishers( id
) ON DELETE CASCADE")
   -> 0.3440s
   -> 1.0160s
== CreateBooksAndAuthorsBooks: migrated (1.2970s) =============================
```

You should see all commands run without any errors. If you connect to MySQL with the command-line client, you can see the two new tables that were created by the migration:

```
$ mysql -uemporium -phacked
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.20-community

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use emporium_development;
Database changed
mysql> show tables;
+-------------------------------+
| Tables_in_emporium_development |
+-------------------------------+
| authors                       |
| authors_books                 |
| books                         |
| publishers                    |
| schema_info                   |
+-------------------------------+
5 rows in set (0.08 sec)
```

The `schema_info` table is where ActiveRecord stores the current version of the database schema, as explained in the previous chapter. Running `select * from schema_info;` should print `3`, which is the current version of our database schema.

■ **Tip** If you want to go back to a previous version of your database model, just specify the version number as a parameter to the migrate script, as in `rake migrate VERSION=0`.

Creating the Book Model

With the database in place, you can now create the ActiveRecord model for books. Following the ActiveRecord naming conventions, we name it `Book`, and create it using the `script/generate` command.

```
$ script/generate model Book --skip-migration
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/book.rb
      create  test/unit/book_test.rb
      create  test/fixtures/books.yml
```

■ **Note** We skipped the creation of the migration file because we already created it in the previous section. Normally, you would create the migration and the model at the same time with the command `script/generate model Book`, but in this case, we wanted a more descriptive name for the migration file.

The script creates the model, plus a unit test and fixture that you can use in your tests.

ActiveRecord Mapping

In the previous chapter, you created the table used for storing authors, and in this chapter, you have created three more: `publishers`, `books`, and `authors_books`. Figure 3-5 shows the entity relationship diagram (ERD) for the Emporium database, which we show to George to hopefully help him see how these tables work better than an Excel spreadsheet. The ERD shows the different relationships between the tables in the database (the 1 indicates the one record part of the relationship and the * represents the many records part), which contain one-to-many, many-to-one, and many-to-many relationships. Before we modify the generated models, let's take a brief look at how to set up these relationships with ActiveRecord.

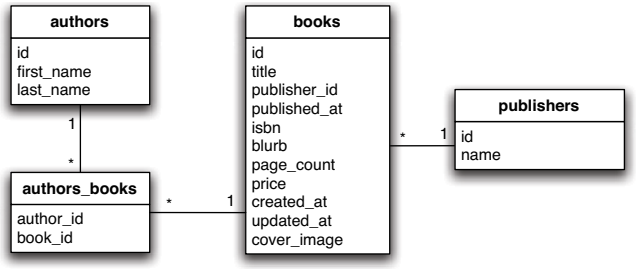


Figure 3-5. *Entity relationship diagram showing the current Emporium database*

One-to-Many Relationship

A *one-to-many relationship* is required when you have one record that owns a set of related records. In Emporium, we have a one-to-many relationship between the `publishers` and `books` tables (see Figure 3-5), because a publisher can have one or more books.

With ActiveRecord, one-to-many relationships are implemented using the `has_many` mapping. Adding a `has_many :books` declaration to the `Publisher` model will inject the methods listed in Table 3-1 into the `Publisher` model.

■ **Note** We list only part of the ActiveRecord API for associations. For a full list of methods, see the Ruby on Rails documentation at `http://rubyonrails.org/api/classes/ActiveRecord/Associations/ClassMethods.html`.

Table 3-1. *Some Methods Introduced by has_many Mapping*

Method	Description
`publisher.books`	Returns an array containing all books associated with the publisher. An empty array is returned if no books belong to the publisher.
`publisher.books << Book.create(...)`	Adds a book to the publisher and sets up the necessary foreign keys.
`publisher.books.delete(some_book)`	Deletes a book from a publisher's collection of books.
`publisher.books = new_books`	Replaces the publisher's collection of books.
`publisher.books_singular_ids=[1,2,3,4]`	Replaces the publisher's books collection with a new collection containing the books having the `id`s 1, 2, 3, and 4.

Method	Description
`publisher.books.clear`	Removes the books from the publisher's collection. The behavior can be configured so that the books are deleted, instead of just removed from the publisher. See the Ruby on Rails documentation on the `:dependent` parameter for more information.
`publisher.books.empty?`	Returns true if the books collection is empty.
`publisher.books.size`	Returns the number of books associated with the publisher.
`publisher.books.find`	Finds an associated object according to the same rules as `ActiveRecord::Base.find`: `http://api.rubyonrails.org/classes/ActiveRecord/Base.html#M000860` .

Many-to-One Relationship

In the previous section, we showed you how to map the `Publisher` model's side of the publisher-book relationship by using a one-to-many relationship. Looking from the `Book` model's perspective, this is a *many-to-one* relationship, since a particular book can have only one publisher.

Many-to-one relationships are implemented using the `belongs_to` ActiveRecord mapping. The `belongs_to :publisher` declaration injects the methods listed in Table 3-2 into the `Book` model. This gives you access to methods such as `book.publisher.nil?`.

Table 3-2. *Some Methods Introduced by belongs_to Mapping*

Method	Description
`book.publisher`	Returns the publisher object or nil.
`book.publisher = new_publisher`	Sets the book publisher to the specified publisher and sets up the required link between the database tables.
`book.publisher.nil?`	Returns true if the book's publisher has not been set.

Many-to-Many Relationship

A *many-to-many relationship* is used when you have two tables that both contain a set of records that can refer to another set of records in the other table. In our case, authors can be associated with one or more books, and books can be authored by one or more authors, so a many-to-many relationship exists between the `authors` and `books` tables. Many-to-many relationships are more complex than one-to-one relationships, as they involve one extra table, referred to as the *join table*. The join table is used for setting up a link between the `authors` and `books` tables.

Many-to-many relationships are set up in ActiveRecord using the `has_and_belongs_to_many` mapping, also referred to as `habtm`. Adding a `has_and_belongs_to_many :books` declaration to the `Author` model will inject the methods listed in Table 3-3.

Table 3-3. *Some Methods Introduced by has_and_belongs_to_many Mapping*

Method	Description
`author.books`	Returns an array of books belonging to this author or an empty array if none have been associated.
`author.books << Book.create(...)`	Adds a book to the author's collection of books and sets up the necessary link in the database by inserting a record in the `authors_books` join table.
`author.books.delete(some_book)`	Removes a book from the author's collection of books. Also removes the corresponding record from the `authors_books` join table.
`author.books = new_books`	Replaces the collection of books with a new one.
`author.books_singular_ids=[1,2,3,4]`	Replaces the author's collection of books with the books having the specified `ids`.
`author.books.clear`	Removes all books from the author's collection and the corresponding row in the `authors_books` join table.
`author.books.empty?`	Returns true if the collection of books is empty.
`author.books.size`	Returns the number of books.
`author.books.find(id)`	Finds the book that is in the author's collection of books and has the specified `id`.

One-to-One Relationship

A *one-to-one relationship* is useful, for example, when you have a master entity that consists of two logically separate entities. For example, a customer can have both a shipping address and billing address. If you don't want to store all of the information in one table, you could put the addresses into separate tables and use a one-to-one relationship to map them to the customer. One-to-one relationships are not used in the Emporium database at the moment.

■ **Tip** Refer to Wikipedia's entry on database normalization, `http://en.wikipedia.org/wiki/Database_normalization`, for more information about how to organize data in a relational database and when to split entities into different tables.

ActiveRecord uses the `has_one` mapping to implement one-to-one relationships. Adding a `has_one :address` declaration to the `Author` model would inject the methods listed in Table 3-4 into the model.

Table 3-4. *Some Methods Introduced by has_one Mapping*

Method	Description
`author.address`	Returns the author's address object or nil.
`author.address = new_address`	Sets the authors address to the specified new address.
`author.address.nil?`	Returns true if the address object hasn't been set.

Modifying the Generated Models

Now that you understand both the database schema and the way ActiveRecord maps to the schema, let's modify the generated models.

Adding the has_many Mapping to the Publisher Model

As we just explained, the one-to-many relationship between the `Publisher` and `Book` model is set up in ActiveRecord by adding the `has_many` declaration to `app/models/publisher.rb`, as highlighted in the following code snippet:

```
class Publisher < ActiveRecord::Base
  has_many :books

  validates_length_of :name, :in => 2..255
  validates_uniqueness_of :name
end
```

This gives you access to, for example, the `books.empty?` method:

```
Publisher.find_by_name('Apress').books.empty?
```

In case you are wondering, `find_by_name` is a dynamic finder, which dynamically (at runtime) creates a SQL query that returns the Apress publisher, or nil if the publisher is not found.

DYNAMIC FINDERS

Dynamic finders are features of ActiveRecord that allow you to use the ActiveRecord API instead of SQL to find objects. Dynamic finders use the `find_by` format and are created by ActiveRecord on the fly at runtime when calling the following, for example:

```
Publisher.find_by_name("Apress")
```

As another example, you could make this call:

```
book.find_all_by_title_and_page_count('Drinking Tequila for Dummies', 538)
```

This dynamically creates a SQL query that finds all books having both the specified title and page count.

Dynamic finders can also create a new record if the query returns no results. This is useful for implementing one-liners like this:

```
Publisher.find_or_create_by_name('Apress')
```

This example creates the publisher `Apress` (if it doesn't already exist) and then returns it.

Adding the belongs_to Mapping to the Book Model

As you learned in the previous section about ActiveRecord mappings, the many-to-one relationship between the `Book` and `Publisher` model is set up with ActiveRecord by using `belongs_to`. Change `app/models/book.rb` as shown here:

```
class Book < ActiveRecord::Base
  belongs_to :publisher
end
```

The `belongs_to` allows you to access, for example, the name of the publisher from the `Book` model:

```
Book.find_by_title('Elvis Peanut Butter Sandwich Recipes 5th
Edition').publisher.name
```

Adding the habtm Mapping to the Book and Author Models

Next, for the many-to-many mapping between the authors and books, add the `has_and_belongs_to_many` mapping to `app/models/book.rb` as follows:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher
end
```

■ **Note** ActiveRecord tries to guess the name of the join table, `authors_books`, by combining the two table names. In our example, ActiveRecord will look for a table named `authors_books`, not `books_authors`, since the string `authors` comes before `books` when compared in lexical order.

We also want to be able to access the books from the author's side of the relationship, so change `app/models/author.rb` as follows:

```
class Author < ActiveRecord::Base
  has_and_belongs_to_many :books

  validates_presence_of :first_name, :last_name

  def name
    "#{first_name} #{last_name}"
  end
end
```

That takes care of the ActiveRecord mappings, but we also want to make sure only valid books are stored in the database. This can be done with validations, which we introduced in Chapter 2.

Adding Validations to the Book Model

Before writing unit tests for the model, you should add some validations to the model. The `Book` model has quite a few attributes that should be validated, as listed in Table 3-5.

Table 3-5. *Validations on the Book Model*

Field	Description
`title`	The title should be at least 1 character long and have a maximum of 255 characters. This validation is done by adding `validates_length_of :title, :in => 1..255` to the model.
`publisher`	A publisher should be assigned to the book. This validation is done by adding `validates_presence_of :publisher` to the model.
`authors`	A book should have at least one author. This validation is done by adding `validates_presence_of :authors` to the model.
`published_at`	The published date should be specified. This validation is done by adding `validates_presence_of :published_at` to the model.
`isbn`	The ISBN number should be in the correct format. This validation is done by adding `validates_format_of :isbn, :with => /[0-9\-xX]{13}/` to the model. The validation uses a regular expression, which checks that there are 13 characters in the ISBN. Note that this is not a complete validation of an ISBN number, but sufficient for our requirements.
`page_count`	The page count should be an integer. This validation is done by adding `validates_numericality_of :page_count, :only_integer => true` to the model.
`price`	The price should be a number. This validation is done by adding `validates_numericality_of :price` to the model.

Next, add each of these validations to the book model, `app/models/book.rb`, as shown here:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  validates_length_of :title, :in => 1..255
  validates_presence_of :publisher
  validates_presence_of :authors
  validates_presence_of :published_at
  validates_numericality_of :page_count, :only_integer => true
  validates_numericality_of :price
  validates_format_of :isbn, :with => /[0-9\-xX]{13}/
  validates_uniqueness_of :isbn
end
```

Cloning the Database

There's one important step left to do before we start writing unit tests: we need to clone the development database to the test environment. Your unit tests will use the test database,

emporium_test, but it hasn't been updated to the latest version. The easiest way of cloning the database structure from the development to the test database is by executing the following command:

```
rake db:test:clone_structure
```

■ **Note** An alternative way of updating the test database is to recreate the database from scratch using migrations, by executing the rake command without specifying any parameters. This first runs all the migrations, and then executes the tests in the test directory.

This is a built-in task that copies the database schema from the emporium_development to the emporium_test database. If you skip this step, you'll get the following error when running the unit test:

```
ActiveRecord::StatementInvalid: Mysql::Error: Table 'emporium_test.books' doesn't
exist: DELETE FROM books
```

Unit Testing Validations

You want to be absolutely sure that the validations are working. One way of doing this is to create a unit test that tests that all fields are validated correctly.

Scaffolding already created a unit test for you, but it contains only a dummy test, so replace the code in test/unit/book_test.rb with the following code:

```
require File.dirname(__FILE__) + '/../test_helper'

class BookTest < Test::Unit::TestCase
  def test_failing_create
    book = Book.new
    assert_equal false, book.save

    assert_equal 7, book.errors.size
    assert book.errors.on(:title)
    assert book.errors.on(:publisher)
    assert book.errors.on(:authors)
    assert book.errors.on(:published_at)
    assert book.errors.on(:isbn)
    assert book.errors.on(:page_count)
    assert book.errors.on(:price)
  end
end
```

The unit test creates a new book without specifying any values for any of the validated fields, such as the price. It then tries to save the book to the database. This triggers a validation of each

of the fields to which you added validations. When you run the test, there is a validation failure on each of the fields—eight in total. We test for this condition with the `assert_equal` method. We also check that validations are done on the correct fields, by calling `book.errors.on` on each validated field. This method returns the actual error message for the specified field, and if it returns nil, we know the validation is not working.

Run the unit test, and you should see it pass without errors:

```
$ ruby test/unit/book_test.rb
```

```
Loaded suite test/unit/book_test
Started
.
Finished in 0.172 seconds.

1 tests, 9 assertions, 0 failures, 0 errors
```

You should also test that a valid book can be saved to the database, by adding the `test_create` method to the unit test, as follows:

```
  fixtures :authors, :publishers

  def test_create
    book = Book.new(
      :title => 'Ruby for Toddlers',
      :publisher_id => Publisher.find(1).id,
      :published_at => Time.now,
      :authors => Author.find(:all),
      :isbn => '123-123-123-1',
      :blurb => 'The best book since "Bodo Bär zu Hause"',
      :page_count => 12,
      :price => 40.4
    )

    assert book.save
  end
```

This test looks up an author and a publisher from the database. These are inserted by the publisher and author fixtures, which have also been added. The test creates a new book with valid parameters, including a publisher and author, and then validates that the book was saved successfully. Recall that calling `save` on the book object returns `true` if there are no validation errors.

Unit Testing the ActiveRecord Mappings

You want to be sure that the mapping between authors, books, and publishers works. George won't be happy at all, if there's a bug in the code that prevents him from adding the latest bestsellers to the catalog.

Adding Fixtures for Books and Publishers

We'll verify that the mapping works by creating unit tests for the mapping code. But let's first add some useful data to the `books` and `publishers` fixture files, which we'll use in later tests.

Open `test/fixtures/books.yml` and add the following two books:

```
pro_rails_ecommerce:
  id: 1
  title: Pro Rails E-Commerce
  publisher_id: 1
  isbn: 199-199-199-1
  published_at: <%= Time.now.strftime("%Y-%m-%d") %>
pro_rails_ecommerce_2:
  id: 2
  title: Pro Rails E-Commerce 2nd Edition
  publisher_id: 1
  isbn: 199-199-199-2
  published_at: <%= Time.now.strftime("%Y-%m-%d") %>
```

Note that the `publisher_id` column has been added to the fixture. This is a reference to a row in the database, which is inserted by the `publishers.yml` fixture file. Currently, no publisher has an `id` equal to 1, so you'll need to add the data to the `publishers` fixture file to complete the mapping.

■ **Tip** You can write ERB in fixtures in the same way as in views. This allows you to create dynamic fixtures, as demonstrated in the `books.yml` fixture file, where `Time.now` is used to generate the `published_at` value. Although dynamic fixtures are useful in some situations, they should generally be avoided as they make tests more complex and less predictable.

Next, add a publisher to the `test/fixtures/publishers.yml` file:

```
apress:
  id: 1
  name: Apress
emporium:
  id: 2
  name: Emporium
```

Recall that you specify the fixtures that the unit test should load by adding a `fixtures` declaration. A couple of tests that we will implement later in this chapter depend on the `authors`, `publishers`, and `books` fixtures. The `books` fixture has not been added to the test yet, so change the `fixtures` line in the unit test as follows:

```
fixtures :authors, :publishers, :books
```

Unit Testing the One-to-Many Mapping

Now we'll put the data into use and verify that we can access a collection of books from a publisher. This is done by adding the new test, `test_has_many_and_belongs_to_mapping`, to the `test/unit/book_test.rb` unit test:

```
def test_has_many_and_belongs_to_mapping
  apress = Publisher.find_by_name("Apress")
  assert_equal 2, apress.books.size

  book = Book.new(
    :title => 'Rails E-Commerce 3nd Edition',
    :authors => [Author.find_by_first_name_and_last_name('Christian', 'Hellsten'),
                 Author.find_by_first_name_and_last_name('Jarkko', 'Laine')],
    :published_at => Time.now,
    :isbn => '123-123-123-x',
    :blurb => 'E-Commerce on Rails',
    :page_count => 300,
    :price => 30.5
  )

  apress.books << book

  apress.reload
  book.reload

  assert_equal 3, apress.books.size
  assert_equal 'Apress', book.publisher.name
end
```

■ **Note** The unit test doesn't call `book.save` explicitly. ActiveRecord is smart enough to know that it must persist the book to the database when the book is added to the author's collection of books. Also note that you could use `assert_difference` (introduced in the previous chapter), instead of two calls to `assert_equal`.

The unit test performs the following tasks in order:

1. Look up a publisher and verify that there are two books in the books collection. These two books are inserted by the fixture at the start of the test.
2. Create a new book and associate two authors with it.
3. Add the new book to the publisher's collection of books.
4. Reload the book and publisher data from the database.
5. Verify that the publisher has three books, instead of the original count of two.
6. Verify that the publisher's name is the one we assigned.

■ **Note** The order the fixtures are listed in is important. The fixture data is inserted in the order it is listed. For example, putting the `publishers` fixture after the `books` fixture would result in a foreign key error when the test is run and Rails tries to insert the fixture data: `ActiveRecord::StatementInvalid: Mysql::Error: Cannot add or update a child row: a foreign key constraint fails`.

Next, run the unit tests. You should see all tests pass without any errors.

```
$ ruby test/unit/book_test.rb
```

```
Loaded suite test/unit/book_test
Started
...
Finished in 0.359 seconds.

3 tests, 13 assertions, 0 failures, 0 errors
```

To see the SQL that is executed by ActiveRecord behind the scenes, tail the `logs/test.log` file by executing the following command in a separate console window:

```
$ tail -f logs/test.log
```

This will monitor the log for changes and print them out to the screen.

Run the unit tests again. You should see the following output from the `test_has_many_and_belongs_to_mapping` test that you just implemented:

```
SQL (0.000000)  BEGIN
  Publisher Columns (0.000000)   SHOW FIELDS FROM publishers
Publisher Load (0.016000)  SELECT * FROM publishers WHERE (publishers.`name` = '
  Book Columns (0.000000)   SHOW FIELDS FROM books
SQL (0.000000)  SELECT count(*) AS count_all FROM books WHERE (books.publisher_i
  Author Columns (0.015000)   SHOW FIELDS FROM authors
Author Load (0.000000)  SELECT * FROM authors WHERE (authors.`first_name` = 'Joe
  Author Load (0.000000)   SELECT * FROM authors WHERE (authors.`first_name` = '
Book Load (0.000000)  SELECT * FROM books WHERE (books.publisher_id = 1)
  SQL (0.000000)   INSERT INTO books (`isbn`, `updated_at`, `page_count`, `price
authors_books Columns (0.016000)  SHOW FIELDS FROM authors_books
  SQL (0.000000)   INSERT INTO authors_books (`author_id`, `book_id`) VALUES (1,
authors_books Columns (0.000000)  SHOW FIELDS FROM authors_books
  SQL (0.000000)   INSERT INTO authors_books (`author_id`, `book_id`) VALUES (2,
Publisher Load (0.000000)  SELECT * FROM publishers WHERE (publishers.id = 1) LI
  Book Load (0.000000)   SELECT * FROM books WHERE (books.id = 9) LIMIT 1
SQL (0.000000)  SELECT count(*) AS count_all FROM books WHERE (books.publisher_i
  Join Table Columns (0.015000)   SHOW FIELDS FROM authors_books
Author Load (0.000000)  SELECT * FROM authors INNER JOIN authors_books ON author
  SQL (0.329000)   ROLLBACK
```

As you can see from the first and last line in the sample output, each test is wrapped in a transaction, and changes done by the test are rolled back at the end of the test.

Adding a Fixture for the Many-to-Many Relationship

Next, we'll add a fixture that contains the data needed in the `authors_books` join table. We will use the data in the next section when writing a unit test that tests the many-to-many mapping. Create a new file named `test/fixtures/authors_books.yml` and add the following code:

```
pro_rails_ecommerce_1:
  author_id: 1
  book_id: 1
pro_rails_ecommerce_2:
  author_id: 2
  book_id: 1
```

The fixture links the two authors defined in the `authors` fixture to a record found in the `books` fixture.

Unit Testing the Many-to-Many Mapping

Change the `fixtures` declaration in the `app/test/unit/book_test.rb` file to use the new fixture:

```
fixtures :publishers, :authors, :books, :authors_books
```

Next, implement a test that verifies that the many-to-many mapping works. This test will verify that you can access the list of authors of a specific book by calling `book.authors`, and that you, from the author's perspective, are able to access the list of books an author has written by calling `author.books`. Open `test/unit/book_test.rb` and add the following method to the end of the file.

```
  def test_has_and_belongs_to_many_authors_mapping
    book = Book.new(
      :title => 'Rails E-Commerce 3nd Edition',
      :publisher => Publisher.find_by_name('Apress'),
      :authors => [Author.find_by_first_name_and_last_name('Christian', 'Hellsten'),
                   Author.find_by_first_name_and_last_name('Jarkko', 'Laine')],
      :published_at => Time.now,
      :isbn => '123-123-123-x',
      :blurb => 'E-Commerce on Rails',
      :page_count => 300,
      :price => 30.5
    )

    assert book.save

    book.reload

    assert_equal 2, book.authors.size
    assert_equal 2, ➥
Author.find_by_first_name_and_last_name('Christian', 'Hellsten').books.size
  end
```

The unit test performs the following steps:

1. Create a new book and assign two authors and one publisher to it.
2. Reload the book from the database.
3. Verify that the book has two authors.
4. Verify that one of the authors has two books, of which one is created by the test and the other by the fixture.

Now run the unit tests:

```
$ ruby test/unit/book_test.rb
```

```
Loaded suite test/unit/book_test
Started
..
Finished in 0.421 seconds.

4 tests, 16 assertions, 0 failures, 0 errors
```

You should see no errors.

Generating Book Administration Code with the Scaffolding Script

With both the database schema and ActiveRecord model in place, we are now ready to start implementing the front-end. The requirements for book administration include five user stories: Add Book, Upload Book Cover, View Book, Edit Book, and Delete Book.

■ **Tip** It's good practice to run all your tests—including unit, integration, and functional—after you make any big changes, as we have done in this chapter. This can be done by running the `rake` command without specifying any parameters. However, at this point, it will throw an error. You can fix this by adding the line `config.active_record.schema_format = :sql` to `config/environment.rb`.

As with the publisher administration interface, we use scaffolding to create the controller, model, and view files by executing the `generate` script:

```
$ script/generate scaffold Book 'admin/book'
```

```
      exists  app/controllers/admin
      exists  app/helpers/admin
      create  app/views/admin/book
      exists  test/functional/admin
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
        skip    app/models/book.rb
        skip    test/unit/book_test.rb
        skip    test/fixtures/books.yml
      create  app/views/admin/book/_form.rhtml
      create  app/views/admin/book/list.rhtml
      create  app/views/admin/book/show.rhtml
      create  app/views/admin/book/new.rhtml
      create  app/views/admin/book/edit.rhtml
      create  app/controllers/admin/book_controller.rb
      create  test/functional/admin/book_controller_test.rb
      create  app/helpers/admin/book_helper.rb
      create  app/views/layouts/book.rhtml
      create  public/stylesheets/scaffold.css
```

You can delete the `public/stylesheets/scaffold.css` file, because you already have a style sheet. Note that the generated functional test will fail if you execute it. You can decide whether to keep it or delete it, but we deleted it by executing the following command:

```
$ rm test/functional/admin/book_controller_test.rb
```

Now we'll introduce you to integration tests, which we'll use to test the front-end instead of functional tests.

Integration Testing

As we mentioned in Chapter 2, Ruby on Rails 1.1 introduced the concept of integration tests. Integration tests can be used to write tests that span multiple controllers and exercise the whole application, from the dispatcher to the database.

Suppose that we wanted to write a test for the whole Emporium administration interface. The test would need to simulate the actions of one or more administrators: logging in to the application; administering authors, publishers, and books; and logging out. Integration tests are a good way of simulating these actions, as they can be used to ensure that related functionality works as expected when multiple controllers and actions are called in sequence.

Another benefit of using integration tests is that they allow you to open multiple sessions, unlike functional tests, which use the same session for the whole test. Opening a new session is done by calling the `open_session` method, which returns a new `session` object. This opens up a whole new range of possibilities for testing your code. For example, you can simulate multiple users accessing the same application at the same time, and you can test for bugs that are related to the session. The `open_session` method also enables you to extend the session with your own methods. This technique can be used to write a domain-specific language (DSL).

■ **Tip** It's a good idea to try to create test cases that are based on the user story. Try to follow the same flow of actions and events as in the user story. Also test alternative use case flows, which might involve invalid user input, for example.

Jamis Buck, one of the Rails core team members, points out on his blog `jamis.jamisbuck.org` that one of the biggest benefits of using integration tests is that you can easily create DSLs. As discussed in Chapter 2, Rails itself uses DSLs for many tasks, such as ActiveRecord mappings and validations. A DSL, as its name implies, is a language you write in Ruby code (or other programming language) for a specific domain. In the context of the Emporium project, for example, one domain is the integration testing of our book administration interface. DSLs should support actions related to a domain.

The following is an example of a DSL. If you read it line by line, you can get a sense of what it does, even without knowing too much about DSLs.

```
require "#{File.dirname(__FILE__)}/../test_helper"

class DSLTest < ActionController::IntegrationTest

  def test_browse_book_store
    george = new_session
    bob = new_session

    george.add_book(...)
    bob.view_book(...)
    bob.add_book_to_cart(...)
  end
```

```
  private

  module TestingDSL
    def add_book(...)
      ...
    end

    def view_book(...)
      ...
    end

    def add_book_to_cart(...)
      ...
    end

    def new_session
      open_session do |sess|
        sess.extend(TestingDSL)
        yield sess if block_given?
      end
    end
  end
end
```

You'll see how to implement integration testing in the following sections, as we complete each of the book administration user stories for this sprint.

Completing the Add Book User Story

As you have noticed, throughout this chapter, we haven't followed TDD very strictly. Instead, we first created the code using scaffolding. Although we can add, list, view, edit, and delete books, the functionality is not tested and we are not confident that it is working as George desires. We'll have to talk to George to find out what exactly should be implemented.

We call George over to our cubicle, which happens to be the only cubicle in the office, reserved exclusively for consultants. George tells us that when adding a book to the system, he must be able to enter all details of the book, including title, price, ISBN, blurb, and so on. Furthermore, George tells us that the current system is difficult to use. Because of this, he has to consult his computer-literate nephew, who enters the details of new books into the system. The blurb text is what is causing him most troubles. When displayed on the website, the blurb text must be nicely styled with, for example, proper headings, bulleted lists, and italicized text, so that it looks as good as possible. On the Web, this requires HTML skills, and because George doesn't know HTML, he can't write the blurb himself. Luckily, there's a simple answer to the problem called Textile.

Textile is a simple text markup language that can be used to write content for the Web without needing to know HTML. RedCloth is a Ruby module that adds Textile support to Rails applications. You'll see how this works when we implement the View Book user story, later in the chapter.

Our first task is to create the integration test we will use to test the book administration implementation.

Creating an Integration Test

We'll create a DSL that will closely match the actions performed in the book administration user stories. Create the integration test by executing the following command:

```
$ script/generate integration_test book
```

```
      exists  test/integration/
      create  test/integration/book_test.rb
```

As with unit tests, the integration test contains only a dummy test, so modify the `test/integration/book_test.rb` file as shown in Listing 3-2.

RUBY BLOCKS

A *block* is a piece of Ruby code that can be passed to a Ruby method. Unlike normal parameters, blocks can be passed to all Ruby methods without explicitly declaring that the method takes a block as a parameter. The method receiving the block, as a parameter, can evaluate the code, by calling the `yield` method.

The following example shows how Ruby blocks can be used for preprocessing and postprocessing by passing a block to the `log` method.

```
def log
  puts "before"
  yield
  puts "after"
end

log { puts "in between" } # block on one line
```

The following is the output of executing this example:

```
before
in between
after
```

The following syntax is preferred for blocks that span more than one line:

```
log do
  calculate_x
  calculate_y
end
```

Listing 3-2. *First Version of Integration Test for the Book Administration Interface*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BookTest < ActionController::IntegrationTest
  fixtures :publishers, :authors
  def test_book_administration
    publisher = Publisher.create(:name => 'Books for Dummies')
    author = Author.create(:first_name => 'Bodo', :last_name => 'Bär')

    george = new_session_as(:george)
    ruby_for_dummies = george.add_book :book => {
      :title => 'Ruby for Dummies',
      :publisher_id => publisher.id,
      :author_ids => [author.id],
      :published_at => Time.now,
      :isbn => '123-123-123-X',
      :blurb => 'The best book released since "Eating for Dummies"',
      :page_count => 123,
      :price => 40.4
    }
  end

  private

  module BookTestDSL
    attr_writer :name

    def add_book(parameters)
      post "/admin/book/create", parameters
      assert_response :redirect
      follow_redirect!
      assert_response :success
      assert_template "admin/book/list"
      assert_tag :tag => 'td', :content => parameters[:book][:title]
      return Book.find_by_title(parameters[:book][:title])
    end
  end

  def new_session_as(name)
    open_session do |session|
      session.extend(BookTestDSL)
      session.name = name
      yield session if block_given?
    end
  end

end
```

Note that the `test_book_administration` test will be used for verifying that the whole book administration works from end to end. The first step in doing this is implementing a test for the Add Book user story.

Also note that the method `new_session_as(name)` is used to open a new session for a virtual user. Inside the method, we use some Ruby magic to extend the new `session` object at runtime with our book-testing DSL. This is done with the `extend` method, which simply adds the instance methods in the `BookTestDSL` module to the `session` object.

We also save the name of the user in an instance variable inside the DSL module. This allows you to use it later, if required.

The line `yield session if block_given?` is used to pass the new session to a block, if a block has been specified.

The integration test performs the following actions, which verify that the Add Book user story works:

1. Create a new author and publisher.
2. Open a new session as George.
3. Create a new book by calling the `create` action with valid parameters.
4. Verify that there is a redirection to the list books view, which should happen if the book was created successfully.

Run the integration test, and you should see that all tests pass:

```
$ ruby test/integration/book_test.rb
```

```
Loaded suite test/integration/book_test
Started
.
Finished in 0.453 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

Since the test didn't fail, you could be tricked into believing that we have just finished the implementation of the Add Book user story, but you can see that this is not the case by opening a browser and going to `http://localhost:3000/admin/book/new`. You should see the front-end for the Add Book user story, as shown in Figure 3-6.

The page you see on the screen was created by the scaffolding script, and includes drop-down lists for the `published_at`, `created_at`, and `updated_at` fields. The values for `created_at` and `updated_at` are generated by Rails automatically, so George shouldn't have to see them. There's also no way of specifying the authors or publisher of the book.

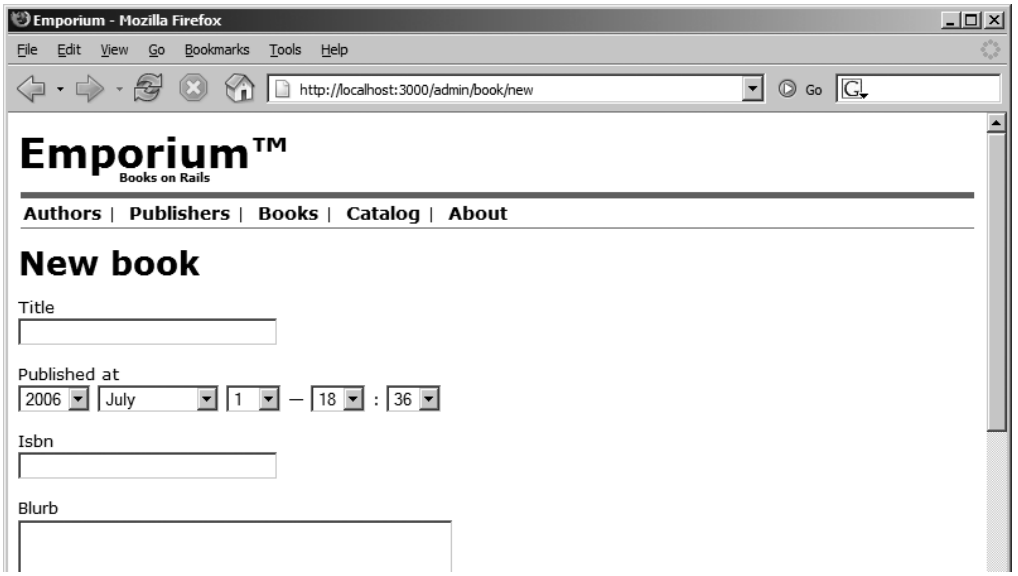


Figure 3-6. *Testing the Add Book user story*

Changing the Controller

The add book page should provide a way of selecting the publisher and one or more authors for the book. This will be done using a drop-down list of all available publishers and a multiple-selection list showing all authors. To be able to show the authors and publishers in the view, we must tell the controller to load all publishers and authors, and pass them to the view.

Open app/controllers/admin/book_controller.rb and add the lines shown in bold to the new and create actions.

```
def new
  load_data
  @book = Book.new
end

def create
  @book = Book.new(params[:book])
  if @book.save
    flash[:notice] = 'Book was successfully created.'
    redirect_to :action => 'list'
  else
    load_data
    render :action => 'new'
  end
end
```

Also, add the following method to the end of the controller:

```
private

def load_data
  @authors = Author.find(:all)
  @publishers = Publisher.find(:all)
end
```

Note that the new method load_data should be declared private, as it is used only internally by the controller. This method loads all authors and publishers from the database and stores them as instance variables, which allows you to access them from the view.

We also needed to change the create action, since it will render the new action's view, which expects to find the authors and publishers, if there is a validation error on the Book object.

The form shown on the add book page is generated by the app/views/admin/book/_form.rhtml file. This file is shared by both the edit and add book pages.

Changing the View

Next, we will use the `collection_select` view helper to generate the drop-down list for publishers. The format for the `collection_select` helper is as follows:

```
<%= collection_select :book, :publisher_id, @publishers, :id, :name %>
```

The first and second parameters tell the helper to which model and attribute to bind the field. The third parameter is used to pass a list of publishers that should be shown in the drop-down list. The two last parameters, `:id` and `:name`, are used to specify that the value for the drop-down list should be the publisher's `id` and that the label should be the publisher's name.

`select_tag` is used for generating a list of authors from which George can select one or more authors. The format for this helper is as follows:

```
<%= select_tag 'book[author_ids][]',
      options_from_collection_for_select(@authors, :id, :name, ➥
@book.authors.collect{|author| author.id}),
        { :multiple => true, :size => 5 }
%>
```

`select_tag` has the following parameters:

- The first parameter, `book[author_ids][]`, specifies to which attribute the field should be bound. Note that the parameter must end with `[]` so that Rails knows that the attribute it needs to bind the value to is an array.
- The second parameter, `options_from_collection_for_select`, is used for generating the list of options that should be shown in the list. This method's first parameter is a collection of authors. The method's second and third parameters, `:id` and `:name`, specify the attributes that should be used as the value and label, respectively. The fourth parameter is used for preselecting the authors that have been assigned to the book.
- The third parameter of the `select_tag` is used for specifying options. In this case, we specify that the list should support multiple-selections and that five authors should be shown.

Next, change the app/views/admin/book/_form.rhtml file as follows:

```
<%= error_messages_for 'book' %>

<p><label for="book_title">Title</label><br/>
<%= text_field 'book', 'title'  %></p>

<p><label for="book_publisher">Publisher</label><br/>
<%= collection_select :book, :publisher_id, @publishers, :id, :name %></p>

<p><label for="book[author_ids][]">Authors</label><br/>
<%= select_tag 'book[author_ids][]',
      options_from_collection_for_select(@authors, :id, :name, ➥
@book.authors.collect{|author| author.id}),
        { :multiple => true, :size => 5 }
%>
</p>

<p><label for="book_published_at">Published at</label><br/>
<%= datetime_select 'book', 'published_at'  %></p>

<p><label for="book_isbn">Isbn</label><br/>
<%= text_field 'book', 'isbn'  %></p>

<p><label for="book_blurb">Blurb</label><br/>
<%= text_area 'book', 'blurb'  %></p>

<p><label for="book_price">Price</label><br/>
<%= text_field 'book', 'price'  %></p>

<p><label for="book_price">Page count</label><br/>
<%= text_field 'book', 'page_count'  %></p>
```

Notice that we use the `text_field`, `collection_select`, `select_tag`, `datetime_select`, and `text_area` helpers for creating the fields. You can test the Add Book user story by first adding a couple of authors and publishers to the database, either through the user interface we created earlier or by executing the following from the console:

```
$ script/console
```

```
Loading development environment.
>> Publisher.create(:name => 'Apress')
>> Author.create(:first_name => 'Salman', :last_name => 'Rushdie')
>> Author.create(:first_name => 'Joel', :last_name => 'Spolsky')
```

Then, open `http://localhost:3000/admin/book/new` in your browser. You should see a page that looks similar to Figure 3-7.

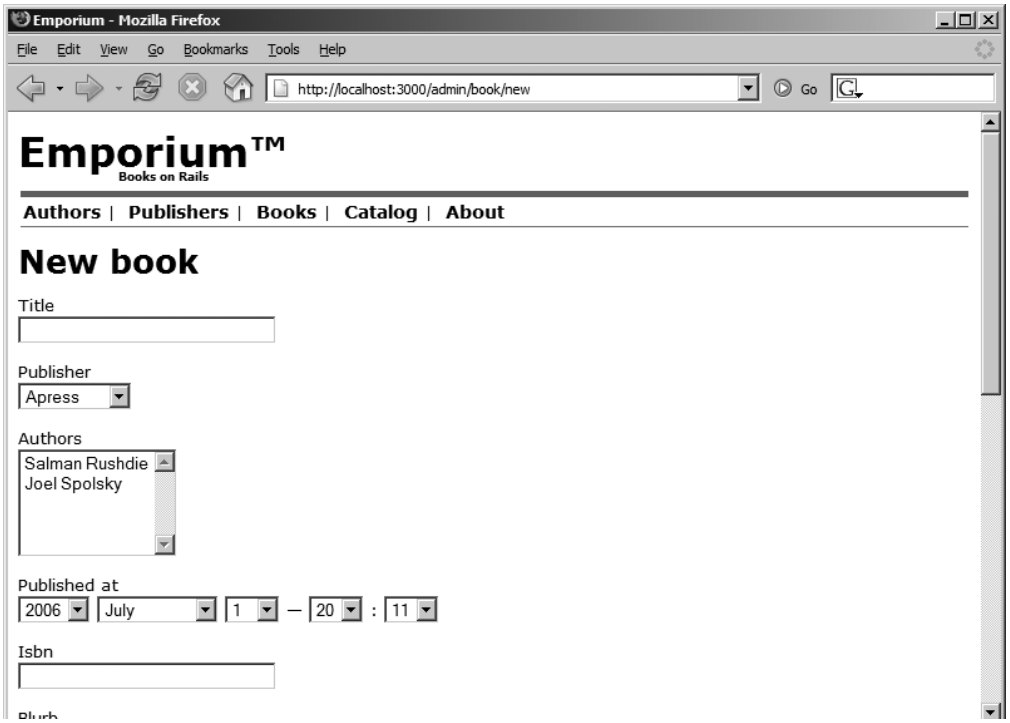


Figure 3-7. *The add book page with publishers and authors*

You should be able to add books to the system by entering all valid information. If you forget to enter something in a required field, you should see validation errors similar to those shown in Figure 3-8.

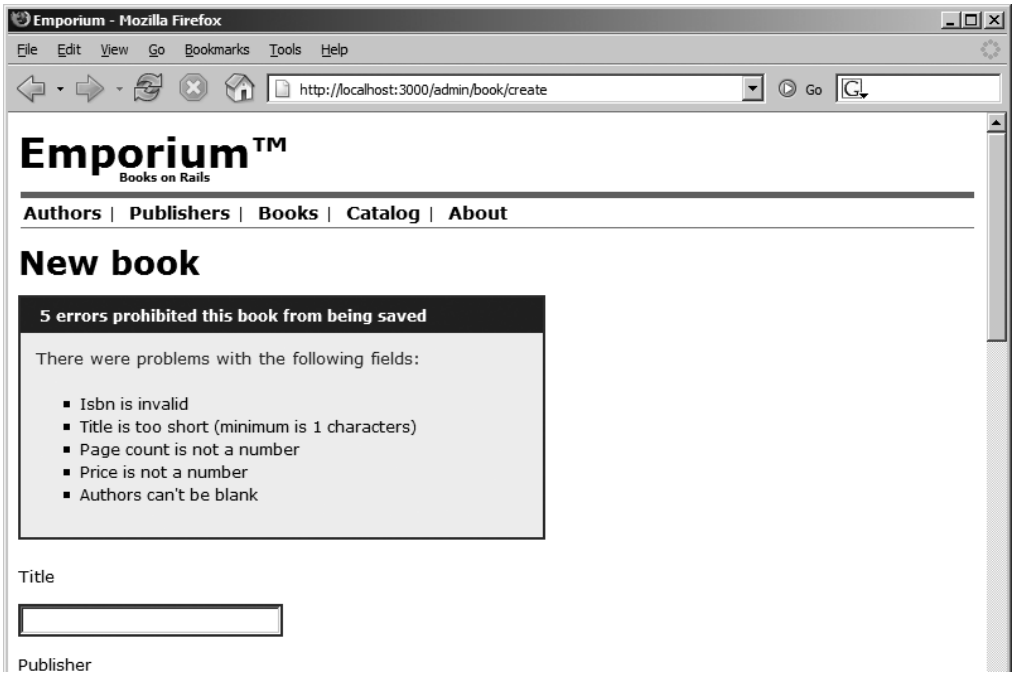


Figure 3-8. *Validation errors*

Updating the Integration Test

As usual, we should change the integration test to reflect the changes we have made to the code. We should test that we can create a book and that the view contains what is expected. Next, change the `add_book` method in the DSL as shown in Listing 3-3.

Listing 3-3. *The Updated Integration Test for the Add Book User Story*

```
def add_book(parameters)
  author = Author.find(:all).first
  publisher = Publisher.find(:all).first

  get "/admin/book/new"
  assert_response :success
  assert_template "admin/book/new"

  assert_tag :tag => 'option', :attributes => { :value => publisher.id }
  assert_tag :tag => 'select', :attributes => {
    :id => 'book[author_ids][]'}

  post "/admin/book/create", parameters
  assert_response :redirect
  follow_redirect!
  assert_response :success
  assert_template "admin/book/list"
  assert_tag :tag => 'td', :content => parameters[:book][:title]
  return Book.find_by_title(parameters[:book][:title])
end
```

Instead of just calling the `create` action with valid parameters and verifying that the request was successful, we now test the complete user story, including that the form contains a list of publishers and authors. This is done with the `assert_tag`, which checks if the drop-down list and multiple-selection list are displayed on the screen.

Run the integration test again, and you should see all tests pass:

```
$ ruby test/integration/book_test.rb
```

```
Loaded suite test/integration/book_test
Started
.
Finished in 0.532 seconds.

1 tests, 8 assertions, 0 failures, 0 errors
```

We have now finished the implementation of the Add Book user story.

Completing the Upload Book Cover User Story

The Upload Book Cover user story is performed by the administrator, George. When adding a book, George should be able to select an image and upload it to the Emporium site. This image is then shown to customers when they are viewing the details of a book.

Adding File Upload Functionality

We don't have to reinvent the wheel to implement file upload functionality. Sebastian Kanthak has already implemented the file upload functionality we need and released it as the FileColumn plugin. The plugin contains view helpers and an extension to ActiveRecord that allows us to implement file upload easily.

Install the FileColumn plugin by executing the following command:

```
$ script/plugin install \
http://opensvn.csie.org/rails_file_column/plugins/file_column/trunk/
```

This downloads the latest version of the plugin from the Internet and installs it in the `vendor/plugins/trunk` directory. After the installation has finished, rename the `trunk` directory to `file_column`. Note that you need to restart WEBrick to activate the plugin.

■ **Tip** For more information about FileColumn, visit `http://www.kanthak.net/opensource/file_column/`. For example, you can discover how to configure FileColumn to resize the uploaded image with the RMagick image processing library.

The FileColumn plugin stores the path to the uploaded image in the database. The exact column where it should store the information is specified with a call to the `file_column` method. Currently, our database schema doesn't contain a column that we can use for this purpose, which is why we'll create it in the next section.

But first, add the `file_column` call to `app/models/book.rb`:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  file_column :cover_image
  validates_length_of :title, :in => 1..255
```

By calling `file_column`, we include the file upload functionality in our model and tell it to store the path to the uploaded image in the `cover_image` column.

■ **Note** At the time of writing, the FileColumn plugin contained an annoying bug that runs a unit test located in the plugin's `lib` directory every time you execute `rake` or a script. To fix this, delete `vendor/plugins/file_column/lib/test_case.rb` and remove the line `require 'test_case'` from the `vendor/plugin/file_column/init.rb` file.

Modifying the Database Schema

We'll use an ActiveRecord migration to add the `cover_image` column to the `books` table. Create the migration with the following command:

```
$ script/generate migration add_book_cover_column
```

```
      exists  db/migrate
      create  db/migrate/004_add_book_cover_column.rb
```

Add the following migration code to `db/migrate/004_add_book_cover_column.rb`.

```
class AddBookCoverColumn < ActiveRecord::Migration
  def self.up
    add_column :books, :cover_image, :string
  end

  def self.down
    remove_column :books, :cover_image
  end
end
```

The migration adds the `cover_image` column to the `books` table, and removes it if we are rolling back changes.

You can now execute the migration with `rake migrate`:

```
$ rake migrate
```

```
(in C:/projects/emporium)
== AddBookCoverColumn: migrating ==============================================
-- add_column(:books, :cover_image, :string)
   -> 0.5150s
== AddBookCoverColumn: migrated (0.5150s) =====================================
```

Cloning the Changes

You should clone the changes to your test database, because we'll create an integration test later in this chapter that tests the file upload functionality. You can clone the development database to test by executing the following command:

```
rake db:test:clone_structure
```

As usual, you could run `rake` without specifying any parameters.

Changing the Form

Next, we'll change the form we created for the Add Book user story so that the user can select an image and upload it. Add the following code to the end of the view `app/views/admin/book/_form.rhtml`.

```
<p><label for="book_cover_image">Cover image</label><br/>
<%= file_column_field 'book', "cover_image" %></p>
```

Note that the file upload functionality requires that we change the form encoding to be `multipart/form-data`. This is done by changing the `start_form_tag` in `app/views/admin/book/new.rhtml`, as follows:

```
<%= start_form_tag( {:action => 'create'}, :multipart => true ) %>
```

You can now test the file upload functionality in your browser by opening `http://localhost:3000/admin/book/new` and selecting a file for the Cover image field. As you can see after clicking the Create button, the path to the uploaded image is stored in the database.

When we implement the View Book user story, we will show you how to use the `url_for_file_column` method to extract the path and display the image on a page:

```
<%= image_tag url_for_file_column(:book, :cover_image) %>
```

■ **Tip** At the time of writing, we couldn't test file uploading with integration tests because of a bug in Rails. But, when it is fixed, you can use the `fixture_file_upload` in your tests to create a valid HTTP parameter that can be used by the `get` and `post` methods; for example, `:cover_image => fixture_file_upload ('/book_cover.gif', 'image/png')`. Note that the `book_cover.gif` image should be in the `fixtures` directory.

Completing the List Books User Story

We already created a page with scaffolding that lists all the books in the system. This page can be accessed at `http://localhost:3000/admin/book/list`. We show it to George and he seems happy, except for two things: he can't sort the list and he only needs to see the publisher's name and the book's title and ISBN.

Changing the View

To fix the book list page, first change the view as follows:

```
<table>
  <tr>
    <th><a href="?sort_by=publisher_id">Publisher</a></th>
    <th><a href="?sort_by=title">Title</a></th>
    <th><a href="?sort_by=isbn">ISBN</a></th>
    <th colspan="3"></th>
  </tr>

<% for book in @books %>
  <tr>
    <td><%=h book.publisher.name %></td>
    <td><%=h book.title %></td>
    <td><%=h book.isbn %></td>
    <td><%= link_to 'Show', :action => 'show', :id => book %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => book %></td>
    <td><%= link_to 'Destroy', { :action => 'destroy', :id => book }, ➥
 :confirm => 'Are you sure?', :post => true %></td>
  </tr>
<% end %>
</table>
<%= link_to 'Previous page', { :page => @book_pages.current.previous } ➥
if @book_pages.current.previous %>
<%= link_to 'Next page', { :page => @book_pages.current.next } ➥
if @book_pages.current.next %>
<br/>
<%= link_to 'New book', :action => 'new' %>
```

The links we added allow George to sort the list when the Publisher, Title, or ISBN column is clicked.

Changing the Controller

The following code implements the sorting. Change the `app/controllers/admin/book_controller.rb` file accordingly.

```
  def list
    @page_title = 'Listing books'
    sort_by = params[:sort_by]
    @book_pages, @books = paginate :books, :order => sort_by, :per_page => 10
  end
```

Note the sort order is specified with the `sort_by` parameter. This parameter is passed to the `paginate` method, which has built-in support for ordering the paginated list.

Adding an Integration Test

We'll update our book administration DSL to include a method for testing the List Books user story. The new method performs a simple smoke test. It accesses the page and verifies that the server responds with an HTTP 200 status code, which means the request was successfully processed.

Change the `BookTestDSL` as follows, adding the code shown in bold.

```
module BookTestDSL
  attr_writer :name

  def list_books
    get "/admin/book/list"
    assert_response :success
    assert_template "admin/book/list"
  end

  def add_book(parameters)
```

Also add the row highlighted below to the end of the `test_book_administration` test. This method simulates George browsing to the book list page, right after he has added a new book.

```
def test_book_administration
  .
  .
  george.list_books
end
```

The finished page can be accessed at `http://localhost:3000/admin/book/list` and should look like Figure 3-9.

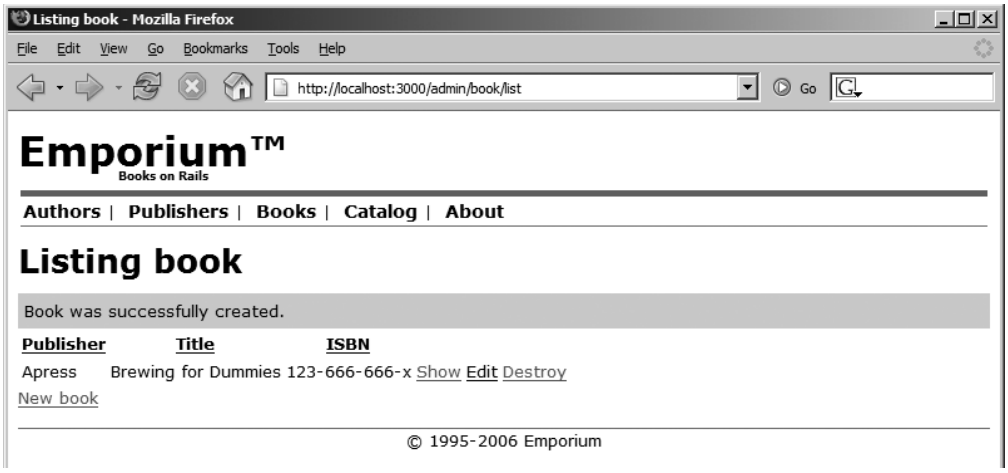


Figure 3-9. *Testing the List Books user story*

Completing the View Book User Story

The View Book user story also needs some cleaning up before George is happy. The code created by the scaffolding displays the values of all database columns directly to the user. This means, for example, that the publisher's ID is shown instead of the publisher's name. We'll fix this and also add code that displays the authors of the book and the book cover.

Changing the View

First, change `app/views/admin/book/show.rhtml` as follows:

```
<dl>
  <dt>Title</dt>
  <dd><%= @book.title %></dd>
  <dt>Publisher</dt>
  <dd><%= @book.publisher.name %></dd>
  <dt>Published at</dt>
  <dd><%= @book.published_at.strftime("%m/%d/%Y at %I:%M%p") %></dd>
  <dt>Authors</dt>
  <dd><%= @book.authors.collect{|author| author.name }.join(', ') %></dd>
  <dt>ISBN</dt>
  <dd><%= @book.isbn %></dd>
  <dt>Blurb</dt>
  <dd><%= textilize @book.blurb %></dd>
  <dt>Price</dt>
  <dd><%= @book.price %></dd>
  <dt>Page count</dt>
  <dd><%= @book.page_count %></dd>
  <dt>Cover image</dt>
  <% if @book.cover_image.nil? %>
  <dd>N/A</dd>
  <% else %>
  <dd><%= image_tag url_for_file_column(:book, :cover_image) %></dd>
  <% end %>
</dl>

<p><%= link_to "Edit", :action => "edit", :id => @book %> |
<%= link_to "Back", :action => "list" %></p>
```

Note that we use `image_tag` and the method `url_for_file_column` to display the uploaded image of the book cover, but only if it exists. We also format the field `published_at` to use a standard format.

Recall that George wanted the Blurb field to be easy to edit. This is why we have used the Textile markup language in the Blurb field, instead of HTML. The Textile markup we entered in the Blurb field is passed through the `textilize` method in the view:

```
<%= textilize @book.blurb %>
```

This translates the Textile markup in the Blurb field to HTML. You'll see this in action in the next section.

■ **Note** The `textilize` method is resource-intensive and should be executed only once (when the object is saved). The resulting HTML should be stored in a database field, for example, `blurb_html`. The conversion can easily be done using a `before_save` filter in the Book model, and then changing the view to display the `blurb_html` column's value, instead of running the conversion for each request.

Changing the Controller

There's one more thing to fix. The view expects to find the instance variable `page_title`, which means you should change the controller's `show` action, as follows:

```
def show
  @book = Book.find(params[:id])
  @page_title = "#{@book.title}"
end
```

You can now access the book details page by clicking the Show link located next to a book on the books list page. Figure 3-10 shows the page after all the changes have been done. Note that the uploaded image is shown at the bottom of the page.

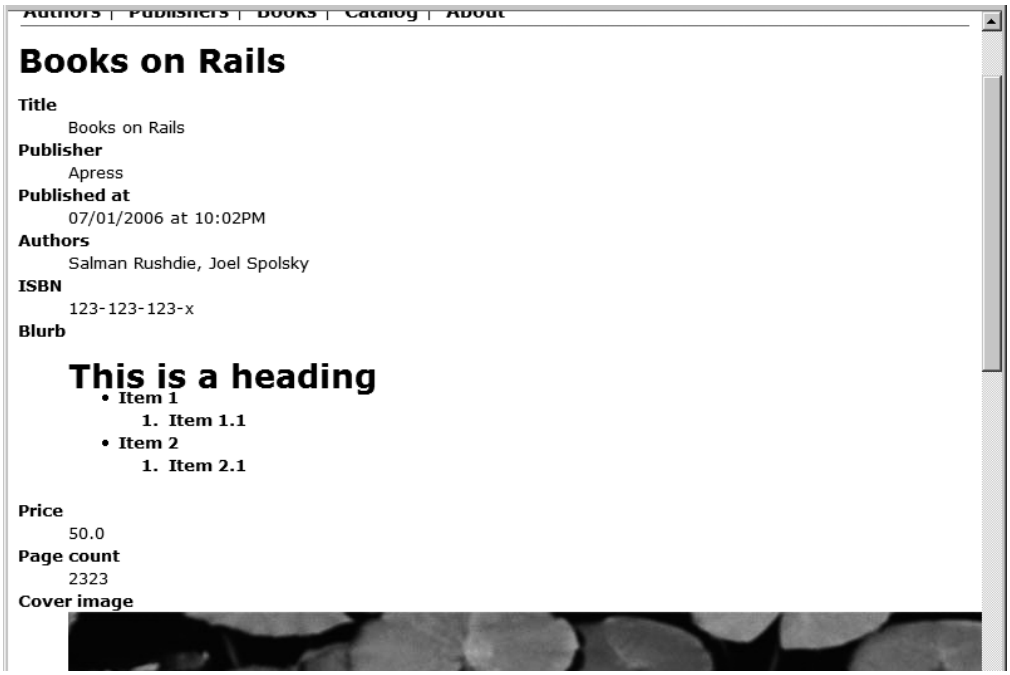


Figure 3-10. *Testing the View Book user story*

Another thing to note about Figure 3-10 is that the Blurb field shows a heading, a bulleted, and a numbered list. In Figure 3-10, we entered the following into the Blurb field:

```
h1. This is a heading

* Item 1
## Item 1.1

* Step 2
## Step 2.1
```

■ **Tip** See `http://en.wikipedia.org/wiki/Textile_(markup_language)` for more information about the Textile markup language.

Adding an Integration Test

We'll also add an integration test for the View Book user story. This is a simple test that verifies that the page doesn't throw an error. Add the following code to the DSL.

```
    def show_book(book)
      get "/admin/book/show/#{book.id}"
      assert_response :success
      assert_template "admin/book/show"
    end
```

The `show_book` method takes a book as a parameter, which it uses to call the `show` action.

Also add the highlighted line, shown in the following code, to the last line of the `test_book_administration` method, right after the line `george.list_books`:

```
  george.list_books
  george.show_book ruby_for_dummies
end
```

This calls the test using the book we created earlier in the test. Run the integration test again to verify that it still passes:

```
$ test/integration/book_test.rb
```

```
Loaded suite test/integration/book_test
Started
.
Finished in 0.531 seconds.

1 tests, 12 assertions, 0 failures, 0 errors
```

Completing the Edit Book User Story

One user story remains for us to implement, before we can call it a day. Luckily, most of the code was generated with scaffolding.

Open a browser and verify that the Edit Book user story works. Add a book to the system and click the Edit link next to the book in the list of books. You should see the following error message:

```
NoMethodError in Admin/book#edit
```

Rails is kind enough to tell us the error is around line 5, which is where we display the list of publishers. The test is failing because we haven't loaded the `publisher` object, which the view expects to find. To fix this, we need to change the action so that it loads the publishers and authors in the same way we did for the Add Book user story. Since we already created the `load_data` method, we only need to add a call to it in the `edit` action, as follows:

```
  def edit
    @page_title = 'Editing book'
    load_data
    @book = Book.find(params[:id])
  end
```

We also need to change the form to use multipart encoding, because we added the file upload functionality earlier in the chapter:

```
<%= start_form_tag( {:action => 'update', :id => @book}, :multipart => true )%>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= link_to 'Show', :action => 'show', :id => @book %> |
<%= link_to 'Back', :action => 'list' %>
```

It's important that you test the edit functionality. Add a new method to the testing DSL:

```
def edit_book(book, parameters)
  get "/admin/book/edit/#{book.id}"
  assert_response :success
  assert_template "admin/book/edit"

  post "/admin/book/update/#{book.id}", parameters
  assert_response :redirect
  follow_redirect!
  assert_response :success
  assert_template "admin/book/show"
end
```

The new `edit_book` method takes an instance of a book as a parameter and the `parameters` hash. The `parameters` hash should contain the new attributes that the book should be updated to use.

Lastly, add a call to the `edit_book` method right after the `george.show_book` line in the `test_book_administration` method:

```
    george.show_book ruby_for_dummies

    george.edit_book(ruby_for_dummies, :book => {
      :title => 'Ruby for Toddlers',
      :publisher_id => publisher.id,
      :author_ids => [author.id],
      :published_at => Time.now,
      :isbn => '123-123-123-X',
      :blurb => 'The best book released since "Eating for Toddlers"',
      :page_count => 123,
      :price => 40.4
    })
end
```

Run the integration test by executing `ruby test/integration/book_test.rb`, and you should see no errors. Verify that you can edit a book by accessing the edit page in your browser.

Testing the Delete Book User Story

The last user story, Delete Book, is already complete. Scaffolding created the `destroy` action in the book controller, which is all we need. But, we can't be sure it works until we have a test in place, so we'll write an integration test for it.

Add the new `delete_book` method to the DSL:

```
def delete_book(book)
  post "/admin/book/destroy/#{book.id}"
  assert_response :redirect
  follow_redirect!
  assert_template "admin/book/list"
end
```

The new method simply calls the `destroy` action and verifies that we are redirected to the list books page.

We'll allow another user, not George, to execute the test, to better illustrate how integration tests can be used. Add the two highlighted lines to the end of the `test_book_administration` method.

```
    :page_count => 123,
    :price => 40.4
  })

  bob = new_session_as(:bob)
  bob.delete_book ruby_for_dummies
end
```

Again, run the tests with `rake test:integrations`. You should see no errors, which means you have successfully implemented the book administration interface.

We quickly do an ad hoc usability test with George, by allowing him to add a couple of books and publishers to the system. He is delighted that everything works and that we could finish it so quickly. We decide to call it a day and head home.

Summary

In this chapter, we introduced you to scaffolding and showed you how to map one-to-many, many-to-one, and many-to-many relationships with ActiveRecord. We also showed you how to write integration tests and use a custom testing DSL for the whole book administration interface. Additionally, you saw how to implement file upload capabilities with the FileColumn plugin and how to use the Textile markup language to simplify content creation. At the end of the chapter, we had a working book inventory management system, with extensive tests that make us confident that we can handle future changes to the system without breaking it.

In the next chapter, we'll implement the front-end for the book catalog functionality, which is what the customer will use.

CHAPTER 4

Book Catalog Browsing

In this chapter, we'll work through setting up the basic functionality of a book catalog from the customer's perspective. We'll build the chapter around four user stories where Jill, George's book-hogging customer, plays the starring role.

For the Emporium book catalog, we will create a simple catalog page for the books, along with pages that display details for individual titles. The interface also will need a way to search for books by their titles and descriptions. We will use Ferret, a full-text search engine written in Ruby, to supply this functionality. Additionally, we will create a latest books page and RSS feed, so that Jill can follow what's new at Emporium.

Getting the Book Catalog Requirements

If there's one person keeping Emporium going, that's Jill. Jill lives just a couple of blocks away from the store. When she rushes through the door with her plasma-TV-sized goggles, George knows that the day is saved.

However, Jill's health is not as it used to be. Her visits have gotten fewer and fewer lately. She would love to support George and buy a lot of new books, but it's just too much effort for her to come over daily. Jill is a smart lady, though, and she's found out that this new thing called the Internet can work as an intermediary between her and her beloved book supply.

To make Jill a happy online customer, George comes up with four user stories for this sprint:

- *Browse books*: Jill needs a way to browse the books in the shop. We will keep the list really simple at this point, just letting her shuffle through the supply and find out about new titles.
- *View book details*: After browsing through titles in the first story or getting a list of matching titles in the second one, Jill needs a way to get specific information about a particular title. As a former librarian, she is obsessed about knowing even the most mundane details of every book she is thinking about buying.
- *Search books*: Sometimes Jill finds out about an interesting topic and wants to know more about the subject. She needs to be able to write a few keywords and get a list of all the titles that match her search.
- *Get latest books*: As a book addict, Jill needs a way to keep current about all new books. She would like to find out about new titles with a single look on the Emporium site. What would make her really happy, however, would be an RSS feed that she could follow on her shiny white iBook without even visiting the website. That would leave her more time for her real pleasure, perusing her precious tomes.

We will tackle these user stories in this chapter, one by one, using the already familiar TDD method.

Implementing the Book Catalog Interface

To be able to really test browsing a list of titles, we need to have a number of books available for viewing. Therefore, we need to expand our `authors.yml`, `publishers.yml`, `books.yml`, and `authors_books.yml` fixture files in `test/fixtures`. You can download the files from the Source Code/Downloads section of `www.apress.com`.

As in the previous chapter, we'll use integration tests for this sprint, because they work well to exercise the book catalog browsing system from end to end. First, we'll create a test stub by using the Rails test generator:

```
$ script/generate integration_test BrowsingAndSearching
```

```
      exists  test/integration/
      create  test/integration/browsing_and_searching_test.rb
```

Again, we'll delete the `test_truth` from the test file and replace it with our real test, as shown in Listing 4-1.

Listing 4-1. *First Version of the Integration Test for the Book Catalog Interface*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BrowsingAndSearchingTest < ActionController::IntegrationTest
  fixtures :publishers, :authors, :books, :authors_books

  def test_browsing_the_site
    jill = enter_site(:jill)
    jill.browse_index
  end

  private

  module BrowsingTestDSL
    attr_writer :name

    def browse_index
      get "/catalog"

      assert_response :success
      assert_template "catalog/index"
      assert_tag      :tag => "dl", :attributes =>
                        { :id => "books" },
                      :children =>
                        { :count => 10, :only =>
                          {:tag => "dt"}}
      assert_tag :tag => "dt", :content => "The Idiot"
    end
  end

  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

In the test case, we create a DSL module, as in the previous chapter. We first test that requesting "/catalog" works and that we get the default index template rendered. Then we check that there is a list of books (laid out with an HTML definition list) on the page and that a book called *The Idiot* exists in that list.

Our test fails magnificently, in plain old TDD way, so it's time to implement the functionality.

First, we need to create the controller for the catalog. We'll name it catalog so that it will match the URL requested in the browse_site test method. Once again, use the familiar script/generate command and give the needed action names (for all our projected user stories) as parameters.

```
$ script/generate controller Catalog index show search latest
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/catalog
      exists  test/functional/
      create  app/controllers/catalog_controller.rb
      create  test/functional/catalog_controller_test.rb
      create  app/helpers/catalog_helper.rb
      create  app/views/catalog/index.rhtml
      create  app/views/catalog/show.rhtml
      create  app/views/catalog/search.rhtml
      create  app/views/catalog/latest.rhtml
```

Implementing the Browse Books User Story

Now that we have the controller in place, we're ready to begin with the Browse Books user story.

Modifying the Controller

At this point, we're interested in only the index action and the corresponding view. Open app/controllers/catalog_controller.rb, which was just created by the generator command. Modify the index method so that it looks as follows:

```
def index
  @page_title = "Book List"
  @book_pages, @books = paginate :books,
                          :per_page => 10,
                          :include => [:authors, :publisher],
                          :order => "books.id desc"
end
```

In the `index` action, we first set the page title so that the layout file will pick it up and show it in the headers of the resulting page. Additionally, the action contains a normal pagination call, just as in Chapter 3. However, this time, we use the `include` parameter for the `paginate` call.

The `include` parameter is used in the ActiveRecord `find` method (which is used internally by `paginate`) to make ActiveRecord build up a join query. This single SQL query will be used not only to find the books, but also to fetch the associated authors and publishers from the database. If we omitted the parameter, our code would end up calling a new SQL query each time we needed to get the author or publisher details for a given book. In our case, it would result in $2n+1$ (where n is the number of books) queries instead of just one. When the site gets more traffic, that could become a huge performance bottleneck.

■ **Note** We can hear you ask, "Where does the $2n+1$ come from?" The first query is the one where all the books are fetched. Then, when we iterate over all the n books and call their `authors` and `publisher` methods, each call will result in an additional SQL query, resulting in two additional queries for each book. The resulting amount of queries is thus 2 queries $\times$ n books + the original query, or $2n+1$.

Modifying the View

Next, open `app/views/catalog/index.rhtml` and replace its contents with the following code.

```
<dl id="books">
  <% for book in @books %>
    <dt><%= book.title %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>


<%= link_to 'Previous page', { :page => @book_pages.current.previous } if ➥
@book_pages.current.previous %>
<%= link_to 'Next page', { :page => @book_pages.current.next } if ➥
@book_pages.current.next %>
```

In the view, we iterate over all the books we got from the controller and show their titles, authors, prices, page counts, and publishers. The `pluralize` helper will show the word "page" in either singular or plural, depending on the value of `book.page_count`. In the end, we show links to next and/or previous page in case there are more than ten books in the `@books` array.

Running the Integration Test

Now that we have our simple browsing functionality implemented, we can run our test case.

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
.
Finished in 0.514885 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

The test passes, but browsing is really not browsing if it involves only a single page. So, let's create another test case that checks that the pagination in our catalog works as expected. Make the following changes to `test/integration/browsing_and_searching_test.rb`:

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BrowsingAndSearchingTest < ActionController::IntegrationTest
  fixtures :publishers, :authors, :books, :authors_books

  def test_browsing_the_site
    jill = enter_site(:jill)
    jill.browse_index
    jill.go_to_second_page
  end

  private

  module BrowsingTestDSL
    attr_writer :name

    def browse_index
      get "/catalog"

      assert_response :success
      assert_template "catalog/index"
      assert_tag      :tag => "dl", :attributes =>
                        { :id => "books" },
                      :children =>
                        { :count => 10, :only =>
                          {:tag => "dt"}}
      assert_tag :tag => "dt", :content => "The Idiot"
    end
```

```
    def go_to_second_page
      get "/catalog?page=2"
      assert_response :success
      assert_template "catalog/index"
      assert_equal    Book.find_by_title("Pro Rails E-Commerce"),
                      assigns(:books).last
    end
  end

  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

In `go_to_second_page`, we first fetch the second catalog page. We then check that we get a normal response and the correct template in return. Finally, we check that the first one of the books in our `books.yml` fixture file is on this page, since the books are ordered in a descending chronological order on the catalog page. Running the tests again confirms that the catalog page is working as expected:

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
.
Finished in 0.110837 seconds.

1 tests, 7 assertions, 0 failures, 0 errors
```

Now that we have a working catalog page, it would be nice to make it the home page of the whole book store. We already briefly mentioned Rails routes in Chapter 2, and now we're going to take advantage of them again. Open `config/routes.rb` and change the line for default root url to look like this:

```
# You can have the root of your site routed by hooking up "
# -- just remember to delete public/index.html.
map.connect ", :controller => "catalog"
```

This means that all the requests for the root url are routed to the default action (`index`) of `CatalogController`.

Implementing the View Book Details User Story

Having a catalog page for a series of books is nice, but it's not suitable for excruciating details about every item. Therefore, the next thing for us to do is to implement a page for individual titles. As always, we start by writing a test for this story.

We already have a test case, so we can just extend that. In `test/integration/browsing_and_searching_test.rb`, we'll add another chapter to the story of Jill, right below `test_browsing_the_site`:

```
def test_getting_details
  jill = enter_site(:jill)
  jill.get_book_details_for "Pride and Prejudice"
end
```

Then we add a new method to our `BrowsingTestDSL` module to keep the test code clean:

```
def get_book_details_for(title)
  @book = Book.find_by_title(title)
  get "/catalog/show/#{@book.id}"
  assert_response :success
  assert_template "catalog/show"

  assert_tag      :tag => "h1",
                  :content => @book.title
  assert_tag      :tag => "h2",
                  :content => "by #{@book.authors.map{|a| a.name}}"
end
```

The `get_book_details_for` method simply fetches a book with the given name from the database, then requests the corresponding `show` page and checks that both the book title and the names of the authors are correctly displayed on the resulting page.

When we created `CatalogController`, we specified that we want to have an action called `show` at hand. Therefore, we already have a stub method `show` in `app/controllers/catalog_controller.rb` and a pretty much empty view file `app/views/catalog/show.rhtml`. Let's now add some flesh around these bones.

Modifying the Controller

Implementing the `show` action in `CatalogController` is a simple two-liner. Add the following to `app/controllers/catalog_controller.rb`:

```
def show
  @book = Book.find(params[:id]) rescue nil
  return render(:text => "Not found", :status => 404)➥
unless @book
  @page_title = @book.title
end
```

All we do is to assign the `@book` instance variable with the book that matches the `id` we get from the browser. If the book is not found, we show a very simple 404 Not Found page. Then we put the title of the book in the `@page_title` instance variable to make it show in the layout.

Modifying the View

In the view file, we'll show the details of the book at hand (remember that the book title is shown by the layout file inside an `h1` element). Add the following to `app/views/catalog/show.rhtml`:

```
<h2>by <%= @book.authors.map{|a| a.name}.join(", ") %></h2>
<%= image_tag url_for_file_column(:book, :cover_image) ➥
    unless @book.cover_image.blank? %>
<dl>
  <dt>Price</dt>
  <dd>$<%= sprintf("%0.2f", @book.price) -%></dd>
  <dt>Page count</dt>
  <dd><%= @book.page_count -%></dd>
  <dt>Publisher</dt>
  <dd><%= @book.publisher.name %></dd>
  <dt>Blurb</dt>
  <dd><%= @book.blurb %></dd>
</dl>

<p><%= link_to "Back to Catalog", :action => "index" %></p>
```

Now the view will show the names of all the authors of a book separated by a comma. We also show the cover image of the book if one has been added, and other details of the book. We run the test again, and see that everything works just fine.

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
..
Finished in 0.231862 seconds.

2 tests, 11 assertions, 0 failures, 0 errors
```

Figure 4-1 shows a book detail page in action.

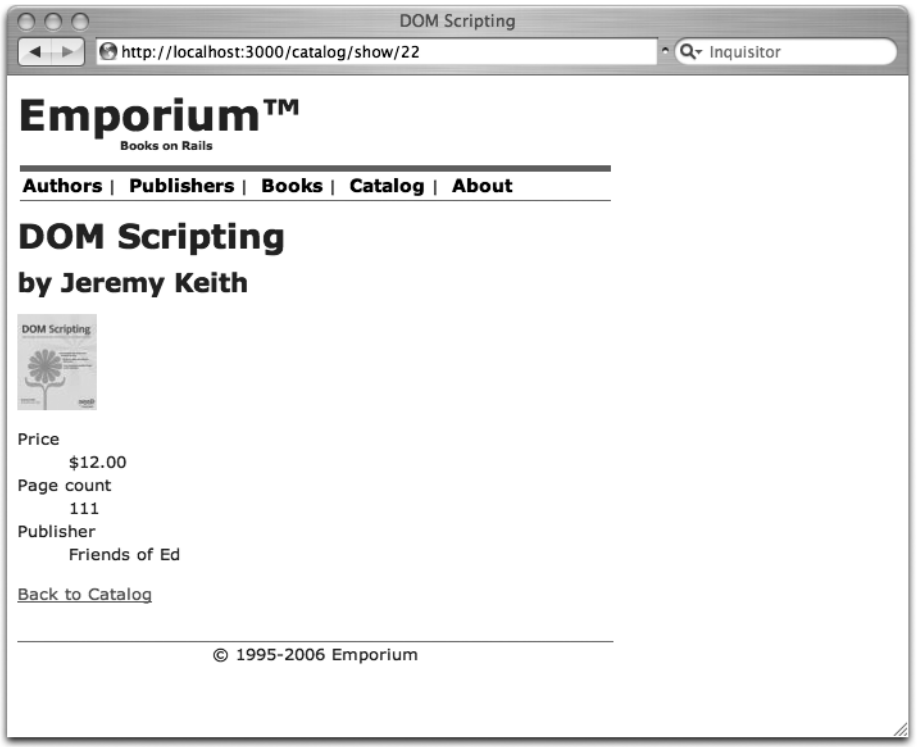


Figure 4-1. *Book detail page*

Adding Links

Now that we have pages for individual books, it would be a good idea to link to them from the catalog list page. Let's make sure that a link exists for each book on the catalog/index page. We create a separate method for checking the links, and then call that method from both the

browse_index and go_to_second_page methods. Add the following new method and calls for it to the BrowsingTestDSL module in test/integration/browsing_and_searching_test.rb:

```
module BrowsingTestDSL
  attr_writer :name

  def browse_index
    get "/catalog"

    assert_response :success
    assert_template "catalog/index"
    assert_tag      :tag => "dl", :attributes =>
                      { :id => "books" },
                    :children =>
                      { :count => 10, :only =>
                        {:tag => "dt"}}
    assert_tag :tag => "dt", :content => "The Idiot"
    check_book_links
  end

  def go_to_second_page
    get "/catalog?page=2"
    assert_response :success
    assert_template "catalog/index"
    assert_equal    Book.find_by_title("Pro Rails E-Commerce"),
                    assigns(:books).last
    check_book_links
  end

  def get_book_details_for(title)
    @book = Book.find_by_title(title)
    get "/catalog/show/#{@book.id}"
    assert_response :success
    assert_template "catalog/show"

    assert_tag      :tag => "h1",
                    :content => @book.title
    assert_tag      :tag => "h2",
                    :content => "by #{@book.authors.map{|a| a.name}}"
  end

  def check_book_links
    for book in assigns(:books)
      assert_tag    :tag => "a", :attributes =>
                      { :href => "/catalog/show/#{book.id}"}
    end
  end
end
```

The next thing to do is to create the links on the index page. Open `app/views/catalog/index.rhtml` and add the highlighted code:

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

Run the tests again. See for yourself the results in Figure 4-2, and bathe in the glory of having implemented yet another user story.

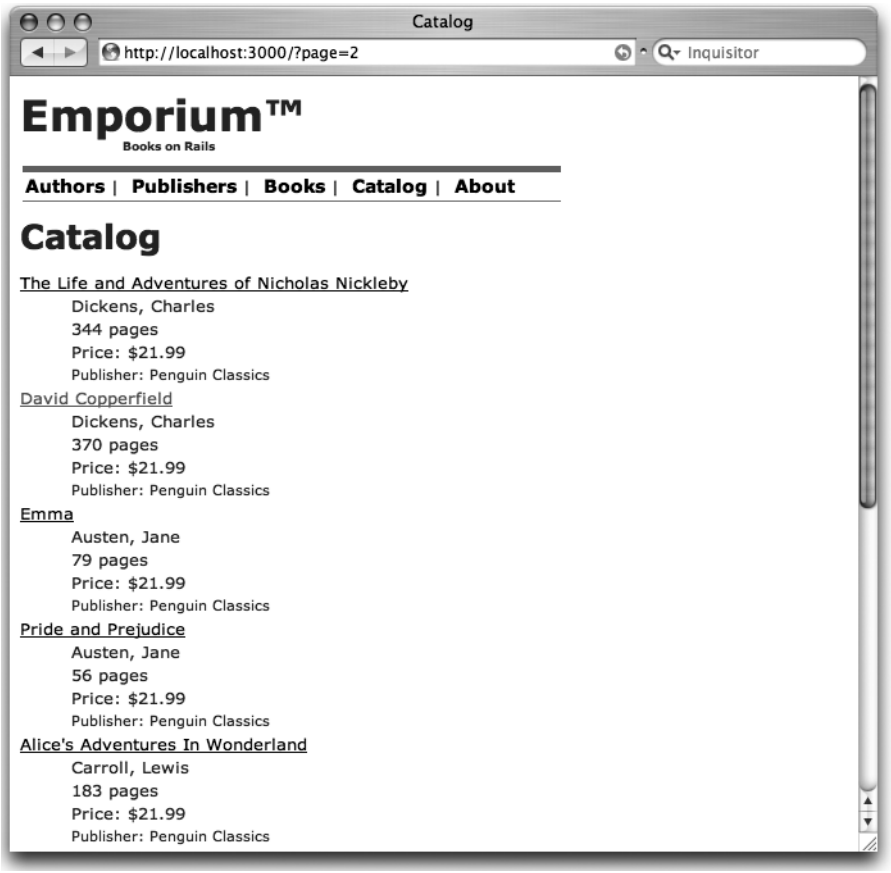


Figure 4-2. *Catalog list page with links*

Implementing the Search Books User Story

An online bookstore, or any other e-commerce site for that matter, would be nothing without search functionality. For simple cases and low loads, it would be enough to just create SQL `SELECT` queries from the search terms to find matching items. However, when the load gets higher and there is more than one table involved in the search, it is worthwhile to use a real full-text search engine for the search. In this chapter, we will use a full-text engine written in Ruby called Ferret (`http://ferret.davebalmain.com/trac`).

Using the Ferret Search Engine

Ferret is open source and uses the MIT license, so it should be a safe choice for any kind of Rails project. There are a couple of other engines available (notably Hyper Estraier and the `acts_as_searchable` Rails plugin that uses it), but we'll use Ferret in this chapter for several reasons:

- Using the `acts_as_ferret` Rails plugin makes integrating Ferret with Rails applications really simple.
- Ferret is a full port of the more famous Java search engine Apache Lucene (`http://lucene.apache.org/`), supporting its whole API. That makes Ferret an easy choice for former Java developers.
- Ferret is reasonably fast, even though it's written in a scripting language. Also, there are versions of Ferret where parts or all of the code are written in C, making it suitable for even the most challenging situations.

Installing Ferret is as easy as a single command:

```
$ sudo gem install ferret
```

The next step is to install the `acts_as_ferret` plugin. We could use Ferret directly, but why duplicate proven and tested code, especially since using the plugin also makes our own code a lot cleaner and less error-prone? You can install the plugin with the normal Rails `plugin` command:

```
$ script/plugin install ➥
svn://projects.jkraemer.net/acts_as_ferret/trunk/plugin/acts_as_ferret
```

```
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/LICENSE
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/rakefile
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/init.rb
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/lib
A    /home/george/projects/emporium/vendor/plugins/ ➥
acts_as_ferret/lib/acts_as_ferret.rb
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/README
Exported revision 59.
```

Now that both Ferret and acts_as_ferret are installed, the only thing we need to make our books searchable is one line in app/models/book.rb:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  acts_as_ferret :fields => [:title, :author_names]
  # lots of omitted code

end
```

With that single line, we have made it possible to do fast searches on books according to their titles and authors. acts_as_ferret now intercepts all create, update, and delete operations of the Book class and updates its full-text index accordingly.

But wait a minute! There is no attribute author_names in the books table. That is correct. Fortunately, acts_as_ferret can index even objects' instance method values, so we'll add a method called author_names to the Book model class. Change app/models/book.rb as shown here:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  acts_as_ferret :fields => [:title, :author_names]
  file_column :cover_image

  validates_length_of :title, :in => 1..255
  validates_presence_of :publisher
  validates_presence_of :authors
  validates_presence_of :published_at
  validates_numericality_of :page_count, :only_integer => true
  validates_numericality_of :price
  validates_format_of :isbn, :with => /[0-9\-xX]{13}/
  validates_uniqueness_of :isbn

  def author_names
    self.authors.map do |a|
      a.name
    end.join(", ") rescue ""
  end
end
```

The author_names method iterates over all of the authors for a given book and returns their names separated by a comma. If there are no authors, it returns an empty string to avoid data type problems in the indexing code.

acts_as_ferret stores its indices in index/[environment] inside your Rails application directory, so your tests won't affect the indices used in development and production. That said, let's create a unit test for the Book class to make sure that the search works correctly. Open test/unit/book_test.rb and paste the following code after the existing tests:

```
def test_ferret
  Book.rebuild_index

  assert Book.find_by_contents("Pride and Prejudice")

  assert_difference Book, :count do
    book = Book.new(:title => 'The Success of Open Source',
              :published_at => Time.now, :page_count => 500,
              :price => 59.99, :isbn => '0-674-01292-5')
    book.authors << Author.create(:first_name => "Steven", :last_name => "Weber")
    book.publisher = Publisher.find(1)
    assert book.valid?
    book.save

    assert_equal 1, Book.find_by_contents("Open Source").size
    assert_equal 1, Book.find_by_contents("Steven Weber").size
  end
end
```

In the beginning of the test, we make sure that the Ferret index is up-to-date. Rails unit tests empty the test database before each test run, but the same doesn't hold true for the index. Therefore it's better to rebuild it so that we can be sure that we always have a similar index before we start running the tests.

Next, we use the class method Book.find_by_contents to search for a book that has "Pride and Prejudice" in either its title or authors. The result should be positive because there is a book with that name in the fixtures we created at the beginning of this chapter.

find_by_contents is a class method created automatically by acts_as_ferret. It is the workhorse of the plugin, taking as its parameters a string of search terms, and returning an array of zero or more objects, just like the normal ActiveRecord find(:all) and find_all_by_* methods.

The last part of the test case tests that a new book is correctly added to the index and is found when searched. We have put this code inside an assert_difference block, just as we did in Chapter 2, to make sure that the book is also saved to the database. We run the test and see that our search engine is working like a dream.

Now that our Book model supports fast search, it's time to implement a search interface for our bookstore.

Updating the Integration Test

We start by extending our integration test to span searching, too. We do this by adding a new method, `searches_for_tolstoy`, to the `BrowsingTestDSL` module in `test/integration/browsing_and_searching_test.rb`, as shown in Listing 4-2.

Listing 4-2. *Test Method for Book Searches*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BrowsingAndSearchingTest < ActionController::IntegrationTest
  fixtures :publishers, :authors, :books, :authors_books

  def test_browsing_the_site
    jill = enter_site(:jill)
    jill.browse_index
    jill.go_to_second_page
    jill.searches_for_tolstoy
  end

  def test_getting_details
    jill = enter_site(:jill)
    jill.get_book_details_for "Pride and Prejudice"
  end

  private

  module BrowsingTestDSL
    include ERB::Util
    attr_writer :name

    def browse_index
      get "/catalog"

      assert_response :success
      assert_template "catalog/index"
      assert_tag      :tag => "dl", :attributes =>
                        { :id => "books" },
                      :children =>
                        { :count => 10, :only =>
                          {:tag => "dt"}}
      assert_tag :tag => "dt", :content => "The Idiot"
      check_book_links
    end
```

```
  def go_to_second_page
    get "/catalog?page=2"
    assert_response :success
    assert_template "catalog/index"
    assert_equal    Book.find_by_title("Pro Rails E-Commerce"),
                    assigns(:books).last
    check_book_links
  end

  def get_book_details_for(title)
    @book = Book.find_by_title(title)

    get "/catalog/show/#{@book.id}"
    assert_response :success
    assert_template "catalog/show"

    assert_tag      :tag => "h1",
                    :content => @book.title
    assert_tag      :tag => "h2",
                    :content => "by #{@book.authors.map{|a| a.name}}"
  end

  def searches_for_tolstoy
    leo = Author.find_by_first_name_and_last_name("Leo", "Tolstoy")

    get "/catalog/search?q=#{url_encode("Leo Tolstoy")}"
    assert_response :success
    assert_template "catalog/search"

    assert_tag      :tag => "dl", :attributes =>
                      { :id => "books" },
                    :children =>
                      { :count => leo.books.size, :only =>
                        {:tag => "dt"}}

    leo.books.each do |book|
      assert_tag :tag => "dt", :content => book.title
    end
  end

  def check_book_links
    for book in assigns(:books)
      assert_tag    :tag => "a", :attributes =>
                      { :href => "/catalog/show/#{book.id}"}
    end
  end
end
```

```
  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

Our new test method makes Jill search for Leo Tolstoy with a search form and checks that the resulting result list will have exactly as many books as Leo has provided the shop, namely two. We use the `url_encode` method to escape white space from the search string. It is provided by the `ERB::Util` library, so we need to require it at the beginning of our module. Last, we test that the books in the resulting list have the correct names by going through all the books written by Leo and checking that there is a `dt` element containing the book's title.

Creating a Search Form Template

Now that we have the integration test made, we can start implementing the thing for real. We first create a simple search form template. Save the following code in `app/views/catalog/_search_box.rhtml`:

```
<%= form_tag({:action => "search"}, {:method => "get"}) %>
<%= text_field_tag :q %>
<%= submit_tag "Search" %>
<%= end_form_tag %>
```

Saving it as a partial makes it possible for us to easily embed the search form in other pages.

In the code, we create a simple form that points to the `search` action and uses the `get` method. Using `get` instead of `post` will make the query string be a part of the URL. That way, Jill can circulate a link to her search results to all of her friends. Our form has only two elements: a text field `q` and the submit button.

In the actual search template, we display the partial using the `render` method. Save the following line to `app/views/catalog/search.rhtml`:

```
<%= render :partial => "search_box" %>
```

Modifying the Controller

Next, open app/controllers/catalog_controller.rb and implement the search action.

```
def search
  @page_title = "Search"
  if params[:commit] == "Search" || params[:q]
    @books = Book.find_by_contents(params[:q].to_s.upcase)
    unless @books.size > 0
      flash.now[:notice] = "No books found matching your criteria"
    end
  end
end
```

In the search action, we first specify the title for the page. Then we continue in two different directions, depending on whether the search form was already submitted or the search page was just requested normally. We do the separation by checking if either the value of a query parameter commit is "Search" or the query variable q is specified. From the _search.rhtml partial view, q contains the search text that was submitted by the search form.

If our code determines that the form has been submitted, it executes the search using the find_by_contents class method and the query parameter q. Furthermore, if there are no books found with the terms, it sets the flash notice to show a message to the user.

Modifying the View

Now we need to extend our search view so that it shows either the books found or the "Not found" notice. Add the following to app/views/catalog/search.rhtml:

```
<%= render :partial => "search_box" %>

<% if @books %>
<p>Your search "<%= params[:q] %>" produced
<%= pluralize @books.size, "result" %>:</p>
<%= render(:partial => "books") %>
<% end %>
```

If the search was successful, we also tell how many hits there were. We use the pluralize helper to show the number of books, and the word "result" in singular or plural depending on the count. Last, we render a partial to show a list of matching books.

We don't have a partial view called books yet, so we need to create it. In the `index` action, we also showed a list of books, so it is a good place to extract the list. Move the following code from `app/views/catalog/index.rhtml` to `app/views/catalog/_books.rhtml`.

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title.t, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

Now we can just replace the moved code in `index.rhtml` with a similar `render` call that we have in the end of the `search.rhtml` template, and that's it! We have a functioning search form in the bookstore.

If you have already added some books to your development system, you can point your browser to `/catalog/search` on your development site and see the result for yourself, as shown in Figure 4-3. (First, you will need to restart your web server, so it will pick up the introduced Ferret code.)

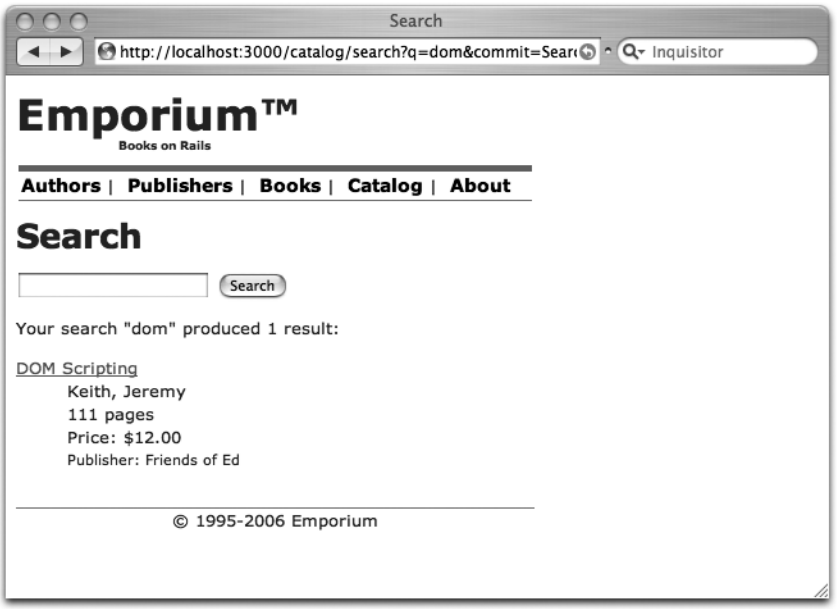


Figure 4-3. *Search interface*

We also need a link to our search functionality, so add the following to `app/views/catalog/index.rhtml`:

```
<p><%= link_to "Search", :action => "search" %></p>

<%= render(:partial => "books") %>

<%= link_to 'Previous page'.t, { :page => @book_pages.current.previous } if ➥
@book_pages.current.previous %>
<%= link_to 'Next page'.t, { :page => @book_pages.current.next } if ➥
@book_pages.current.next %>
```

The search functionality is now implemented

Implementing the Get Latest Books User Story

So far, we have created a book catalog that lets Jill browse and search books, and see their details. The last part of the sprint is to implement the ultimate desire of a book-lover: a list of the latest books. We'll implement this feature both as a normal web page and as an RSS feed, so that Jill can skip the step of using a browser altogether. Again, we'll start by writing a test for the latest books page.

Updating the Integration Test

Add another method to the `BrowsingTestDSL` module in `test/integration/browsing_and_searching_test.rb`:

```
def views_latest_books
  get "/catalog/latest"
  assert_response :success
  assert_template "catalog/latest"

  assert_tag      :tag => "dl", :attributes =>
                    { :id => "books" },
                  :children =>
                    { :count => 10, :only =>
                      {:tag => "dt"}}
  Book.latest.each do |book|
    assert_tag    :tag => "dt", :content => book.title
  end
  check_book_links
end
```

You can see that the method is similar to `browse_index` and `go_to_second_page`, but it has a different URL and desired template. The only thing special here is that we iterate over the `Book` objects returned by `Book.latest` and check that there is a `dt` element for each book. To make this work, we first need to create a `latest` class method for our `Book` class. Add the following method to `app/models/book.rb`:

```
def self.latest
  find :all, :limit => 10, :order => "books.id desc",
             :include => [:authors, :publisher]
end
```

We could have used the `find` method as such in our test. However, we're going to need the exact same code later, so it's a good idea to wrap it inside a class method. We also need to add a call to our new method in the actual test case:

```
def test_browsing_the_site
  jill = enter_site(:jill)
  jill.browse_index
  jill.go_to_second_page
  jill.get_book_details_for "Pride and Prejudice"
  jill.searches_for_tolstoy
  jill.views_latest_books
end
```

Now that we have a (failing, but you guessed that) test in place, the next thing to do is to update the controller.

Modifying the Controller

Open `app/controllers/catalog_controller.rb` and fill the `latest` action with content:

```
def latest
  @page_title = "Latest Books"
  @books = Book.latest
end
```

There's nothing special in there. We just set the page title and then use the `Book.latest` class method we just created to fetch the ten latest books.

Modifying the View

The view file, `app/views/catalog/latest.rhtml`, is even simpler:

```
<%= render :partial => "books" %>
```

We can fire our test case and see that everything works oh so smoothly.

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
..
Finished in 0.478978 seconds.

2 tests, 56 assertions, 0 failures, 0 errors
```

We double-check in the browser to see the page shown in Figure 4-4. Filled with self-confidence, we rush on to the final task of this code sprint.

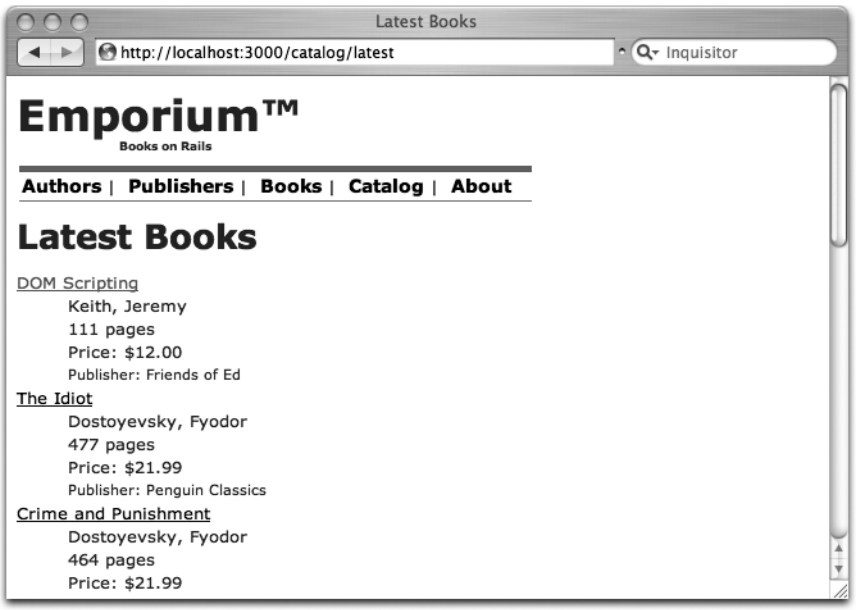


Figure 4-4. *Latest books page*

Creating an RSS Feed

Creating an RSS feed in Rails is painstakingly easy. RSS feeds are essentially just XML files served like normal HTML pages. Rails supports three kinds of template files out of the box. You are already familiar with the HTML templates with the `.rhtml` suffix. The second type is the Builder XML template, with an `.rxml` suffix, which we will use for this case. The third type? You will learn about that in the next chapter, and boy will that be fun. But first we'll create an RSS feed for Jill.

Once more, we'll create another method in our integration test. Add the following method to the `BrowsingTestDSL` module in `test/integration/browsing_and_searching_test.rb`:

```
def reads_rss
  get "/catalog/rss"
  assert_response :success
  assert_template "catalog/rss"
  assert_equal "application/xml", response.headers["type"]

  assert_tag      :tag => "channel",
                  :children =>
                    { :count => 10, :only =>
                      {:tag => "item"}}
  Book.latest.each do |book|
    assert_tag    :tag => "title", :content => book.title
  end
end
```

The method follows the familiar scheme. However, this time we also check that the response type is XML instead of HTML. It is also worth noting that we can use the same `assert_tag` methods here that we used for HTML documents, even though the output is XML.

Just as before, we call our new method from the `test_browsing_the_site` test method:

```
def test_browsing_the_site
  jill = enter_site(:jill)
  jill.browse_index
  jill.go_to_second_page
  jill.get_book_details_for "Pride and Prejudice"
  jill.searches_for_tolstoy
  jill.views_latest_books
  jill.reads_rss
end
```

Implementing the controller method is straightforward. We use the same information as in the `latest` action, so we can call it the same way we call any other method. After that, we just render our action normally, only this time, we don't want to use the layout file (because we're rendering XML).

```
def rss
  latest
  render :layout => false
end
```

The view file is where the most difference between a normal HTML page and a Rails-powered RSS feed lies. This time, we don't use the standard `.rhtml` templates, but rather `.rxml` templates powered by the Builder library. With Builder, XML output is specified using nested code blocks. For our RSS feed, we'll create the `app/views/catalog/rss.rxml` file, as shown in Listing 4-3.

Listing 4-3. *app/views/catalog/rss.rxml*

```
xml.instruct! :xml, :version=>"1.0", :encoding=>"UTF-8"

xml.rss("version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/") do
  xml.channel do
    xml.title @page_title
    xml.link(url_for(:action => "index", :only_path => false))
    xml.language "en-us"
    xml.ttl "40"
    xml.description "Emporium: Books for people"

    for book in @books
      xml.item do
        xml.title(book.title)
        xml.description("#{book.title} by #{book.author_names}")
        xml.pubDate(book.created_at.to_s(:long))
        xml.guid(url_for(:action => "show", :id => book, :only_path => false))
        xml.link(url_for(:action => "show", :id => book, :only_path => false))
      end
    end
  end
end
```

Every code block started by an `xml.tag` command in a Builder template will result in a `<tag>` element in the output. Thus, the output of the code in Listing 4-3 would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>Latest Books</title>
    <link>http://0.0.0.0:3000/catalog</link>
    <language>en-us</language>
    <ttl>40</ttl>
    <description>Emporium: Books for people</description>

    <item>
      <title>The Idiot</title>
      <description>The Idiot by Fyodor Dostoyevsky</description>
      <pubDate>April 26, 2006 20:18</pubDate>
      <guid>http://0.0.0.0:3000/catalog/show/17</guid>
      <link>http://0.0.0.0:3000/catalog/show/17</link>
    </item>

  ... more items ...

  </channel>
</rss>
```

Note that we can use all the normal Rails helper methods, like `url_for`, in `.rxml` templates, just as in normal `.rhtml` views. However, because we're not creating the XML code by hand, we can be sure that the output is always well-formed XML.

Running the integration test reveals that everything works fine. Encouraged, we open `http://localhost:3000/catalog/rss` in a browser that supports RSS feeds (such as Safari on Mac OS X or Firefox on other platforms) and show George how the feed functionality works for Jill, as shown in Figure 4-5. George is excited, and we can pat ourselves on the back. Another sprint is completed.

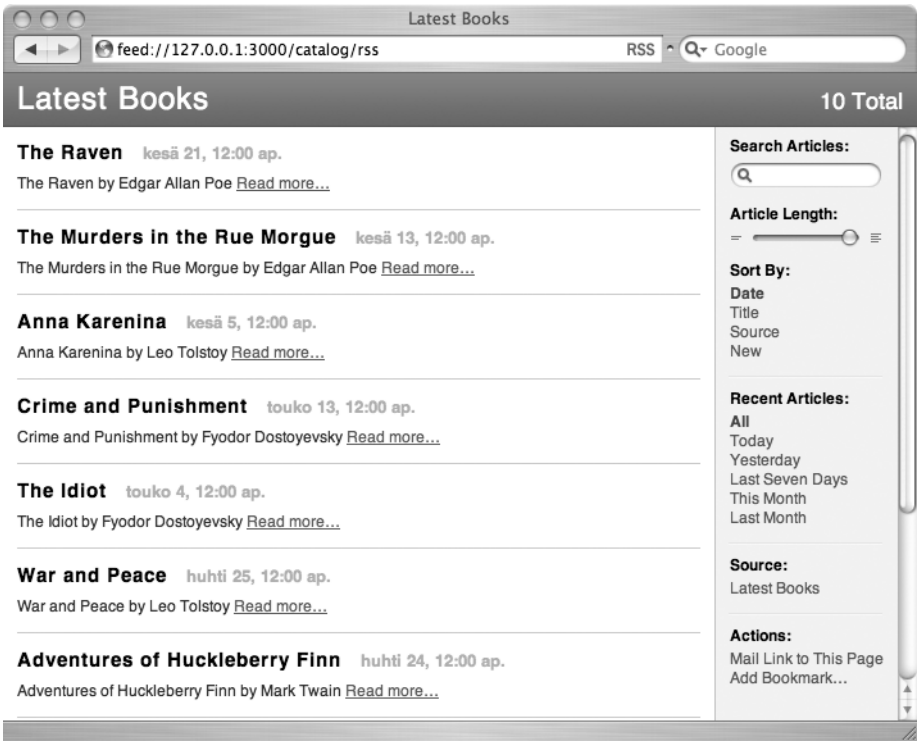


Figure 4-5. *Working RSS feed*

Summary

In this chapter, we implemented the basic functionality of the online bookstore that is visible to a normal user like Jill. This consisted of four user stories: browsing the list of books, searching books, visiting pages for individual books, and seeing lists of latest books in the store in both a web page and an RSS form.

During the course of the chapter, we showed you how to use the `include` parameter in ActiveRecord finder methods to avoid unnecessary SQL queries and use layouts to avoid repeating view code. We also integrated the Ferret full-text search engine with our Rails application using the acts_as_ferret plugin. Finally, we created RSS feeds using Builder XML templates, which saved us a lot of time. In the next chapter we will create a shopping cart for Jill to fill.

CHAPTER 5

Shopping Cart Implementation

In the previous chapter, we built the general interface for Emporium. That's a good start for an online store, but it doesn't bring George any revenues. Now it's time for us to take the next step toward a working e-commerce site: create a shopping cart where customers can drag all the interesting books they find—and preferably a few more.

A shopping cart is also a perfect match for taking advantage of Ajax to provide shoppers with a snappier and more responsive user interface. In this chapter, we will show you how to fully harness the power of Rails Ajax helpers to build a lightning-fast shopping cart, but we'll also make sure that that it works on older browsers that don't support JavaScript.

Getting the Shopping Cart Requirements

Again, Jill, George's faithful customer, plays the main role in our user stories for this sprint:

- *Add items to the cart*: The most important shopping cart feature of any e-commerce application is the ability to add items to the cart. If building up a heaping shopping cart is easy, that will have a positive effect on the cash flow.
- *Remove items from the cart*: No matter how badly George wants Jill to buy all the books she inadvertently added to the cart, that's just not how you make customers happy.
- *Clear the cart*: Sometimes Jill goes totally bonkers. She just can't help adding every book she runs into to the shopping cart. But then she remembers she is on a pension, not on a 100-foot yacht, and just wants to clean up the darn thing. So, we need to make it possible to empty the shopping cart with a single stroke.

The last user story to implement for a shopping cart is check out—the ultimate goal of a web shopping tour. Checking out is a bit more complicated than the other user stories, so we're going to tackle it separately in Chapter 9. There, we'll also cover how to integrate with credit card payment gateways.

Setting Up the Shopping Cart

Once again, we start our sprint by creating a test case for adding an item to the shopping cart. This time, we will use the Rails functional tests. We could use integration tests, but since we are testing the functionality of a single controller (which we will create first) in this case, we agree to use functional tests for this sprint.

Creating the Controller

Our first step is to use the `script/generate` command to create the new `Cart` controller, as follows:

```
$ script/generate controller Cart
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/cart
      exists  test/functional/
      create  app/controllers/cart_controller.rb
      create  test/functional/cart_controller_test.rb
      create  app/helpers/cart_helper.rb
```

Creating a new controller for our shopping cart also provides us with a stub file for the functional tests.

Adding a Functional Test

Open `test/functional/cart_controller_test.rb` and replace the dummy `test_truth` method with the following test case:

```
def test_adding
  assert_difference(CartItem, :count) do
    post :add, :id => 4
  end

  assert_response :redirect
  assert_redirected_to :controller => "catalog"
  assert_equal 1, Cart.find(@request.session[:cart_id]).cart_items.size
end
```

In this test, we first check that posting a form to the `add` action actually creates a new `CartItem`. Then we check that after the form post, we are redirected to the catalog controller and that the current shopping cart is populated with an item.

If you can't figure out what's happening here, don't worry. Once we start implementing the shopping cart, it will be clear how the test works.

Creating the Models

Before we can add anything to a shopping cart, we need to create two models: `Cart` for the shopping carts themselves and `CartItem` for books stored in a cart. It should come as no surprise that we use the `script/generate` command to generate these models:

```
$ script/generate model Cart
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/cart.rb
      create  test/unit/cart_test.rb
      create  test/fixtures/carts.yml
      exists  db/migrate
      create  db/migrate/005_create_carts.rb
```

```
$ script/generate model CartItem
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/cart_item.rb
      create  test/unit/cart_item_test.rb
      create  test/fixtures/cart_items.yml
      exists  db/migrate
      create  db/migrate/006_create_cart_items.rb
```

Next, we need to specify the associations between the new (and some old) models. (ActiveRecord database relationships are covered in Chapter 3.) In our case, the cart items can be seen as a join model between a cart and books. In theory, we could have used a direct `has_and_belongs_to_many` association between carts and books, but the items also need to store some data of their own. One example of such data is the quantity of a given book in a cart. Jill might want to buy one book for herself and one for her nephew Carl. We also need to store the price of a book at the time it was added to the cart so that it won't change, even if the price of a book increased during the user's shopping session.

Therefore, we edit the model files book.rb, cart_item.rb, and cart.rb in app/models to make the associations between carts, cart items, and books look as follows:

```
class Book < ActiveRecord::Base
…
  has_many :cart_items
  has_many :carts, :through => :cart_items
…
end

class CartItem < ActiveRecord::Base
  belongs_to :cart
  belongs_to :book
end

class Cart < ActiveRecord::Base
  has_many :cart_items
  has_many :books, :through => :cart_items
end
```

Note that we use the new has_many :through syntax that was implemented in Rails 1.1. With the new syntax, we can access books belonging to a cart directly by using @cart_object.books, even though the two classes don't have a direct relationship. Without using the new :through option, we would need to laboriously go through the CartItem class: @cart_object.cart_items.map {|ci| ci.book }.

Next, we populate the CreateCartItems migration that was created by the script/generate command. Open 006_create_cart_items.rb and make it look like the following:

```
class CreateCartItems < ActiveRecord::Migration
  def self.up
    create_table :cart_items do |t|
      t.column :book_id, :integer
      t.column :cart_id, :integer
      t.column :price, :float
      t.column :amount, :integer
      t.column :created_at, :datetime
    end
  end

  def self.down
    drop_table :cart_items
  end
end
```

We can leave the `CreateCarts` migration file untouched, since the default content is fine for the moment. Run the migrations with the `rake db:migrate` command.

```
$ rake db:migrate
```

```
(in /home/george/projects/emporium)
== CreateCarts: migrating =====================================================
-- create_table(:carts)
   -> 0.1681s
== CreateCarts: migrated (0.1685s) ============================================

== CreateCartItems: migrating =================================================
-- create_table(:cart_items)
   -> 0.0778s
== CreateCartItems: migrated (0.0781s) ========================================
```

Modifying the Controller

Now that we have our models in place, we need to do something to keep a shopping cart at hand while Jill is browsing the store. The easiest way to do that is to use the filter functionality for Rails controllers—in this case, the `before_filter` macro. Add the following in the beginning of `CartController` (`app/controllers/cart_controller.rb`) and `CatalogController` (`app/controllers/catalog_controller.rb`):

```
before_filter :initialize_cart
```

The filter makes the controller call the `initialize_cart` method before running an action. We could implement the `initialize_cart` method in both `CartController` and `CatalogController`, but since we don't want to repeat ourselves, we put the definition to the `ApplicationController` (in `app/controllers/application.rb`), which is by default the parent class of all our controllers.

```
class ApplicationController < ActionController::Base
  private

  def initialize_cart
    if session[:cart_id]
      @cart = Cart.find(session[:cart_id])
    else
      @cart = Cart.create
      session[:cart_id] = @cart.id
    end
  end
end
```

ACTIONCONTROLLER FILTERS

Filters are a powerful way to control your application logic in your controllers. You can extract common code from actual actions to filters, and then make those filters run for every appropriate action. The `initialize_cart` filter we use for the shopping cart is a good example of filter usage. We want a shopping cart object at hand for every action in `CartController` and `CatalogController`, so we extract the cart initialization code in a `before_filter`.

ActionController offers three types of filters:

- `before_filter`: Appends a method call to the `before_filter` chain of the controller. The chain is executed before any affected action is run, and if any of the filters returns false, the chain is aborted *and the actual action is not run*. Therefore, it's a good way to enforce user authentication, for example, as you will see in Chapter 8.
- `after_filter`: Similar to `before_filter`, except that the `after_filter` chain is executed after the actual action is run. Therefore, it's not possible to abort the action from an `after_filter`.
- `around_filter`: A combination of the two other types of filters. Using this filter, you can maintain state through the action execution. So, for example, you can use an `around_filter` to measure the time spent for an action.

Both `before_filter` and `after_filter` macros can take the actual filter code parameter in three forms: a symbol, a proc object, or a filter class.

If the first parameter for a filter macro call is a symbol, a method with the same name as the symbol is executed as the filter:

```
class CartController < ApplicationController
  before_filter :initialize_cart, :only => [:index, :show]

  private
  def initialize_cart
    # do the magic
  end
end
```

For a quick-and-dirty filter, you can use a proc object as filter parameter. The current controller is automatically passed as a block parameter to the proc.

```
class AdminController < ApplicationController
  before_filter {|controller| false unless controller.current_user }, ➥
:except => :login
end
```

Continued

If you want to reuse your filter code in many different places, it might be worth separating the code in its own class:

```
class LoggingFilter
  def self.filter(controller)
    controller.logger.info "#{controller.request.request_uri} called"
  end
end

class AdminController < ApplicationController
  after_filter LoggingFilter
end
```

When using an `around_filter`, you need to use the class form for the filter. In that case, you define methods `before` and `after` in the filter class, which are then executed before and after the actual action code. Note that you can also use the `:only` and `:except` parameters to restrict the filter calls to only some specific action in a controller.

For more information about ActionController filters, see `http://api.rubyonrails.org/classes/ActionController/Filters/ClassMethods.html`.

In the `initialize_cart` method, we check whether there is an item called `:cart_id` in the `session` hash and if there is, load a cart with that `id`, also fetching all of the cart's items with the same query. If no `cart_id` is stored in the session, we create a new cart and store the `id` of the new cart to the `session` hash. This way, we can be sure that a single session-wide `Cart` object is at hand everywhere in both `CatalogController` and `CartController`. Once we now know that there is always a `Cart` object `@cart` present, we can also show it in the actions of the `CatalogController` we created in the previous chapter.

Creating the Views

We want to create a floating cart that will show all the items we have in our shopping cart and will appear on all the catalog pages. Let's first create a partial called `app/views/cart/_cart.rhtml`:

```
<% if flash[:cart_notice] %>
  <%= render :partial => "cart/cart_notice" %>
<% end %>

  <h3>Your Shopping Cart</h3>
```

```
  <ul>
    <% for item in @cart.cart_items %>
    <li id="cart_item_<%= item.book.id %>">
      <%= render :partial => "cart/item", :object => item %>
    </li>
    <% end %>
  </ul>
  <p id="cart_total"><strong>Total: $<%= sprintf("%0.2f", @cart.total) %> ➥
</strong></p>
```

In the partial, we first show a notice if there is something to notify. Then we show every item in the cart. Last, we show the subtotal of the whole cart, formatted with `sprintf` to always show two decimal places. The `Cart` class doesn't have a `total` method, so let's add it to `app/models/cart.rb`.

```
def total
  cart_items.inject(0) {|sum, n| n.price * n.amount + sum}
end
```

`inject` is a method for arrays and other enumerable objects that can be used to calculate sums, factorials, and so on of all the items in the container object. It takes one initial parameter (in our case, `0`) and passes it as the first block parameter (in our case, `sum`) for the first iteration. Then it passes each item of `cart_items` to the block as `n`, one at a time, updating the `sum` all the time. After it has gone through all of the items, it returns the final value of `sum`.

Although a verbal explanation of `inject` might sound incomprehensible, it is actually fairly easy to use and often makes using explicit loops obsolete. We could, for example, use the following code to get the same result as we do with one line using `inject`:

```
sum = 0
for item in cart_items
  sum =+ item.price * item.amount
end
```

From the `cart/_cart.rhtml` partial, you can see that we call two additional partials, `cart/_item.rhtml` and `cart/_cart_notice.rhtml`. Let's create them at once, too. Create a new file `app/views/cart/_item.rhtml` and add the following to it:

```
<%= link_to item.book.title, :action => "show",
      :controller => "catalog", :id => item.book.id %>
<%= pluralize(item.amount, "pc", "pcs") %>,
$<%= sprintf("%0.2f", item.price * item.amount) %>
```

The `item` partial shows a link to the book details page, the number of this book in the cart, and the total cost for this title. As in Chapter 4, we use the `pluralize` helper to choose the singular or plural form of "pc," depending on the number of items.

The `app/views/cart/_cart_notice.rhtml` is a one-liner, just showing the possible notice indicating a change in the cart:

```
<p id="cart_notice"><%= flash[:cart_notice] %></p>
```

Now that we have a partial to show on the catalog pages, we need to add it to the layout file `app/views/layouts/application.rhtml`, as shown in Listing 5-1. We also add a few JavaScript include tags to get the power of the Prototype and `script.aculo.us` JavaScript libraries that come bundled with Rails (and which we'll need when we start using Ajax in our application, as described in the upcoming "Implementing the Add Items to the Cart User Story" section).

Listing 5-1. *Additions to the Application Layout File*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title><%= @page_title || 'Emporium' %></title>
  <%= stylesheet_link_tag "style" %>

  <%= javascript_include_tag :defaults %>
</head>
<body>

<div id="header">
  <h1 id="logo">Emporium&trade;</h1>
  <h2 id="slogan">Books on Rails</h2>
</div>

<div id="menu">
  <ul>
    <li><a href="/admin/author">Authors</a> | </li>
    <li><a href="/admin/publisher">Publishers</a> | </li>
    <li><a href="/admin/book">Books</a> | </li>
    <li><a href="/">Catalog</a> | </li>
    <li><a href="/about">About</a> </li>
  </ul>
</div>

<div id="content">
  <%= "<h1>#{@page_title}</h1>" if @page_title %>
  <% if flash[:notice] %>
    <div id="notice">
      <%= flash[:notice] %>
    </div>
  <% end %>
  <%= yield %>
</div>
```

```
<% if @cart %>
<div id="shopping_cart">
<%= render :partial => "cart/cart" %>
</div>
<% end %>

<div id="footer">
 &copy; 1995-2006 Emporium
</div>

</body>
</html>
```

We also need to add a link for adding a book to the cart. We do that in the `catalog/_books.rhtml` partial that is used to show every individual book item on the catalog pages:

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd>
        <strong>
          <%= link_to "+",  :controller => "cart",
                            :action => "add", :id => book %>
        </strong>
      </dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

To make our shopping cart float on the catalog pages and look a bit nicer, we need to add some CSS rules to the `style.css` style sheet, as shown in Listing 5-2.

Listing 5-2. *Additions to the Style Sheet*

```
#shopping_cart {
  border-left: 3px solid #666;
  background: #aaa;
  position: fixed;
  bottom: 0;
  right: 0;
  width: 200px;
  height: 100%;
  padding: 5px 10px;
}

#shopping_cart ul,
#shopping_cart li {
  list-style: none;
  margin: 0;
  padding: 0;
}

#shopping_cart h3 {
  padding-top: 4em;
}

#cart_notice {
  border: 2px solid #58A986;
  background: #B2FFD3;
  padding: 3px;
  position: absolute;
  top: 0;
  left: 10px;
}

body {
  background-color: #fff; color: #333;
  margin-right: 230px;
}
```

Notice the use of fixed positioning to make the shopping cart always appear "above the fold." Now you can open the catalog index page in the browser, and it should look like Figure 5-1.

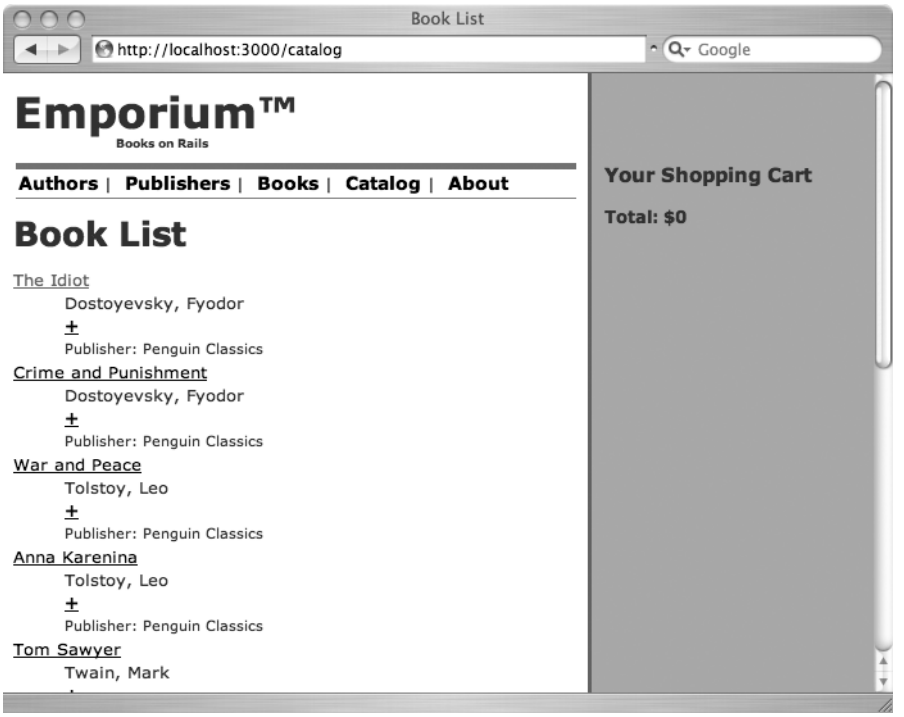


Figure 5-1. *Book catalog page with shopping cart*

Implementing the User Stories

With our shopping cart set up for action, we're ready to start implementing our user stories. Of course, we start with adding items.

Implementing the Add Items to the Cart User Story

We're going to use two approaches to implementing the Add Items to the Cart user story. One is the "traditional" way, which takes care of users whose browser doesn't support JavaScript. The other way will use Ajax techniques to provide a streamlined user interface.

The Traditional Way

We need to implement the action for adding an item to a cart in our `CartController` (in `app/controllers/cart_controller.rb`), as follows:

```
def add
  @book = Book.find(params[:id])

  if request.post?
    @item = @cart.add(params[:id])
    flash[:cart_notice] = "Added <em>#{@item.book.title}</em>"
    redirect_to :controller => "catalog"
  else
    render
  end
end
```

The code checks whether the request was an HTTP `POST` or `GET` request and acts on that information. In the case of a form `POST`, we add the item, insert a notice to the `flash` hash, and redirect to the catalog controller. If the user got to the action by clicking a link, we show her a confirmation page with a form that points to the same action.

Create `app/views/cart/add.rhtml` and add the following to it to implement the confirmation page:

```
<strong>Please confirm adding <em><%= @book.title %></em>
to your shopping cart.</strong>
<%= button_to "Confirm", :action => "add", :id => params[:id] %>
```

`button_to` is a Rails helper that behaves just like `link_to`, but instead of an anchor element, it creates an inline form with a single button. When a user clicks this button, she gets to the same `add` action, this time with a `POST` request, and will get the new item added to the cart. This might sound awkward, but there are two good reasons to make the application behave like this:

- You should never use `GET` requests (normal links) to change the state of a web application. So-called *web accelerators* (or *link fetchers*) often crawl through all the links of a loaded web page and fetch the linked pages while the user is reading the page. If fetching those links would result in adding random items to the shopping cart, we would probably end up with a bunch of less-than-happy customers.

- This traditional way to add items to a shopping cart will be visible to only the few customers who are using a browser that doesn't support JavaScript. In the next section, we will implement an interface that will be used by the vast majority of customers, and that is about gazillion times slicker than the approach described in this section.

Now adding an item to a cart almost works. However, there is no `add` method in the `Cart` class, so we need to add it to `app/models/cart.rb` before we can start filling up the carts.

```
def add(book_id)
  items = cart_items.find_all_by_book_id(book_id)
  book = Book.find(book_id)

  if items.size < 1
    ci = cart_items.create(:book_id => book_id,
                           :amount => 1,
                           :price => book.price)
  else
    ci = items.first
    ci.update_attribute(:amount, ci.amount + 1)
  end
  ci
end
```

The idea behind the `add` method is that if there is already a certain amount of the current book in the cart, we increment that amount by one, using the `update_attribute` method. If there is no sign of the given book, a new item will be added and its amount will be set to `1`.

We're now finished with the old-fashioned way of adding books to the cart. Our test runs fine (try if you don't believe us), so we can extend our application to work with Ajax.

Ajax'ing It

Ajax, dubbed by Jesse James Garrett of Adaptive Path, stands for Asynchronous JavaScript and XML. It's actually not a single technique at all, but a bunch of techniques that can be used together to update a web page without doing a full page refresh. You can use Ajax to implement faster and more interactive user interfaces than could be possible with normal page refreshing. The most famous example of Ajax usage is probably Google Maps (`http://maps.google.com`), where the map is updated without refreshing the whole page when the user drags or zooms the map.

■ **Tip** If you're looking for a sound introduction to Ajax, consider *Foundations of Ajax* by Ryan Asleson and Nathaniel T. Schutta (ISBN 1-59059-582-3).

While Ajax is a very cool and useful addition to the arsenal of any web developer, it doesn't come without problems. To use Ajax-driven sites, a user needs to have a fairly recent browser with JavaScript enabled (JavaScript libraries used in Rails, `script.aculo.us` and Prototype, officially support Internet Explorer 6.0 and up, Mozilla Firefox 1.0/Mozilla 1.7 and higher, and

Apple Safari version 1.2 and up). Even having a JavaScript-capable browser is not a complete assurance of working Ajax. For example, older Opera versions don't support the `XMLHttpRequest` JavaScript object, which is the heart of Ajax.

Therefore, it is advisable to make sure that your application works even without JavaScript and Ajax (unless you're willing to abandon some of your customers). A good way to ensure backward-compatibility is to first make an application work without Ajax—what we just did—and only after that add the groovy Ajax interface.

For testing that adding items to the cart works with Ajax, we add another test to our functional test file `test/functional/cart_controller_test.rb`:

```
def test_adding_with_xhr
  assert_difference(CartItem, :count) do
    xhr :post, :add, :id => 5
  end
  assert_response :success
  assert_equal 1, Cart.find(@request.session[:cart_id]).cart_items.size
end
```

You can see that it's almost identical to the `test_adding` test method, with just two differences:

- Instead of calling the `post` test helper method, we call `xhr` (alias for `xml_http_request`). It simulates an Ajax'ed call to the same `add` action.
- We don't want to be redirected after the action, but instead get a normal 200 HTTP Success response.

To make our `add` controller action work correctly with Ajax calls, we need to modify `app/controllers/cart_controller.rb` slightly:

```
def add
  @book = Book.find(params[:id])

  if request.xhr?
    @item = @cart.add(params[:id])
    flash.now[:cart_notice] = "Added <em>#{@item.book.title}</em>"
    render :action => "add_with_ajax"
  elsif request.post?
    @item = @cart.add(params[:id])
    flash[:cart_notice] = "Added <em>#{@item.book.title}</em>"
    redirect_to session[:return_to] || {:controller => "catalog"}
  else
    render
  end
end
```

With `request.xhr?`, we can check if the request was made by Ajax, just as we can check whether a request was done using `POST` or `GET`. What we do is pretty much the same as with `POST` requests. However, since we want the flash message to be accessible only to the current action (and not the next one), we use the `flash.now` hash instead of just `flash`.

The most important difference is that instead of redirecting, we now render a template called `add_with_ajax`. But we're not rendering a normal `.rhtml` template. Instead, we now use an `.rjs` template. These templates debuted in Rails 1.1 and are a great way to do JavaScript-driven changes to a web page in pure Ruby. Create a new file, `app/views/cart/add_with_ajax.rjs` and add the following code to it:

```
page.replace_html "shopping_cart", :partial => "cart"
page.visual_effect :highlight, "cart_item_#{@item.book.id}", :duration => 3
page.visual_effect :fade, 'cart_notice', :duration => 3
```

`page` is an object provided to the template that represents the web page in question. All the changes done to the page are normally done through it. `page.replace_html` takes as its arguments first the DOM `id` of the element being replaced (here, for example `<div id="shopping_cart">`) and the value it's being replaced with. As you can see, you can use partials there just as with `render` calls, as well as simple strings.

`page.visual_effect` is a method that calls the `script.aculo.us Effect.*` JavaScript methods to do some visual effects for elements in the page. In our case, we highlight the `cart_item` element for a newly created item for three seconds, using the so-called Yellow Fade Technique. In addition, we slowly fade out the notice text element from the page.

■ **Tip** You can read more about the Yellow Fade Technique at `http://www.37signals.com/svn/archives/000558.php`.

Our application is now ready to receive Ajax requests for adding new cart items. However, we need to change the links in our pages to actually use Ajax. Rails makes this extremely easy. All you need to do is to replace a `link_to` call with a similar `link_to_remote` call. However, as we want to maintain the backward-compatibility of our application, we specify the optional `href` attribute for the `link` tag, so that it will behave as a normal link if the users don't have JavaScript enabled or supported in their browser.

Using all the attributes makes the `link_to_remote` call a bit messy, so we create our own helper method for this in `app/helpers/application_helper.rb`. For the sake of brevity, let's create similar helpers for removing a book from the cart and clearing the whole cart at the same time, as shown in Listing 5-3. We will use them in the later sections of this chapter.

Listing 5-3. *Shopping Cart Helpers*

```
module ApplicationHelper
  def add_book_link(text, book)
    link_to_remote text, {:url => {:controller => "cart",
              :action => "add", :id => book}},
              {:title => "Add to Cart",
               :href => url_for( :controller => "cart",
                          :action => "add", :id => book)}
  end

  def remove_book_link(text, book)
    link_to_remote text, {:url => {:controller => "cart",
                        :action => "remove",
                        :id => book}},
                        {:title => "Remove book",
                          :href => url_for( :controller => "cart",
                                      :action => "remove", :id => book)}
  end

  def clear_cart_link(text = "Clear Cart")
    link_to_remote text,
            {:url => { :controller => "cart",
                       :action => "clear" }},
            {:href => url_for(:controller => "cart",
                              :action => "clear")}
  end
end
```

Now, instead of using the whole `link_to_remote` call in our views, we can make our `catalog/_books.rhtml` partial look as clean as this:

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd>
        <strong>
          <%= add_book_link("+", book) %>
        </strong>
      </dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

Drag-and-Drop

Although the Add Items to the Cart user story is now fully functional, there is one thing George still wants us to implement: drag-and-drop shopping. Fortunately, with the functionality that is already in place, adding that capability is very easy.

First, we need to make the books in the catalog pages draggable. Make the highlighted changes to `app/views/catalog/_books.rhtml`:

```
<ul id="books">
  <% for book in @books %>
  <li class="book" id="book_<%= book.id %>">
    <dl>
      <dt><%= link_to book.title.t, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small><%= 'Publisher'.t %>: <%= book.publisher.name %></small></dd>
      <dd>
        <strong>
          <%= add_book_link("+", book) %>
        </strong>
      </dd>
    </dl>
  </li>
  <%= draggable_element("book_#{book.id}", :revert => true) %>
  <% end %>
</ul>
```

We changed the list to an unordered list and gave each book a list element of its own, so that it can be referenced uniquely by its `id` (for example, `book_75`). Then we made each book item a draggable element with the `draggable_element` method. Stating `:revert => true` in the call makes the item slide back to its original position when the mouse button is released during dragging. If you now reload the catalog page in a browser, you can drag the catalog items around the browser window.

To make each item look more like an individual element, we need to style the list a bit. Add the following at the bottom of `public/javascripts/style.css`:

```
#books {
  list-style: none;
  padding: 0;
  float: left;
}
```

```
#books .book {
  float: left;
  border: 4px solid #ccc;
  background-color: #fff;
  margin: 10px;
  padding: 5px;
  cursor: pointer;
}
```

The second part of implementing drag-and-drop functionality in a Rails application is to define an element as a drop-receiving element. This is done using the (surprise!) `drop_receiving_element` helper. In `app/views/layouts/application.rhtml`, add the following code after the `shopping_cart` element:

```
    <div id="shopping_cart">
    <%= render :partial => "cart/cart" %>
    </div>
    <%= drop_receiving_element("shopping_cart", :url =>
        { :controller => "cart", :action => "add" }) %>
```

This makes the `shopping_cart` division act as a receiver for the dragged book items. Whenever a book is released over the shopping cart `div`, an Ajax call to the `add` action of `CartController` is made. You can test this by reloading the catalog page and dragging a book to the cart. The log file should show something like the following:

```
Processing CartController#add (for 127.0.0.1 at 2006-10-09 15:07:07) [POST]
  Session ID: bc2009ee48c083165c6196ac7ff4b44c
  Parameters: {"action"=>"add", "id"=>"book_22", "controller"=>"cart"}
  [4;35;1mCart Load (0.000296) [0m   [0mSELECT * FROM carts WHERE (carts.id = 1654)
LIMIT 1 [0m
  [4;36;1mBook Load (0.000420) [0m   [0;1mSELECT * FROM books WHERE (books.id =
'book_22') LIMIT 1 [0m

ActiveRecord::RecordNotFound (Couldn't find Book with ID=book_22):
```

You can see that the `id` passed to the `add` action is the DOM `id` of the dragged element. However, since the `add` action wants only the actual `id` of the book (`22` in this case), we need to clean up the passed value a bit. Add the highlighted code to `app/controllers/cart_controller.rb`:

```
  def add
    params[:id].gsub!(/book_/, "")
    @book = Book.find(params[:id])
```

`gsub!` is a method that can be used to replace parts of a `String` object in place. It takes two parameters: a regular expression that is to be sought in the string and a replacement string. In our case, we want to remove the `book_` in front of the actual `id` number, so we replace it with an empty string. After the call, `params[:id]` has only the numeric `id` of the book in question and can thus be used in the `Book.find` call on the following line. If the expression is not found in the string—which is the case when the action is called normally by clicking the + link, resulting in `params[:id]` being `22`, for example—the string is left untouched.

That was everything needed to make drag-and-drop work. We're now ready with the first functionality of our shopping cart: adding items to it. Run the functional tests, see them roar through, and finally try out the cart in your browser (see Figure 5-2), basking in the glory of being an Ajax developer. If you want to make sure the system works even without Ajax, turn off JavaScript in your browser and try to add items to your cart the old-fashioned way.

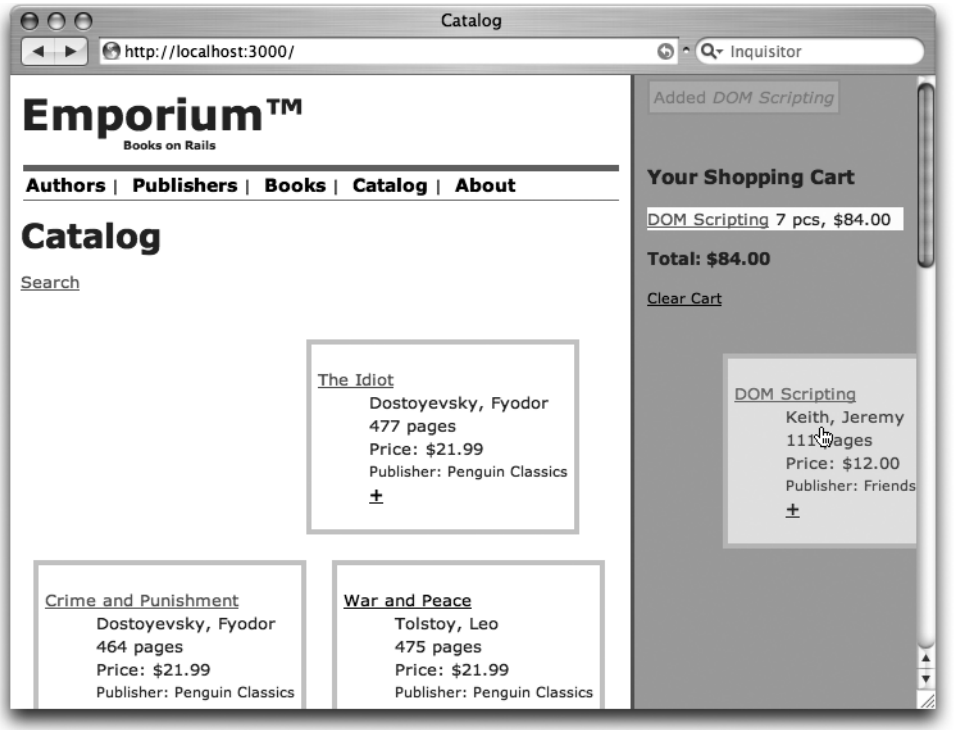


Figure 5-2. *Adding an item to the cart*

Implementing the Remove Items from the Cart User Story

Removing items from the shopping cart is pretty much the inverse action of adding an item. Therefore, we can duplicate much of the code we did for the adding part, with some slight modifications. First, we extend the functional test case in `test/functional/cart_controller_test.rb` to also test for item removal:

```
def test_removing
  post :add, :id => 4
  assert_equal [Book.find(4)], Cart.find(@request.session[:cart_id]).books

  post :remove, :id => 4
  assert_equal [], Cart.find(@request.session[:cart_id]).books
end

def test_removing_with_xhr
  post :add, :id => 4
  assert_equal [Book.find(4)], Cart.find(@request.session[:cart_id]).books

  xhr :post, :remove, :id => 4
  assert_equal [], Cart.find(@request.session[:cart_id]).books
end
```

Just as with the addition, the first test checks that the removal of items works correctly with the traditional way and the second one tests the Ajax functionality.

We start implementing the removal functionality by adding a `remove` method to the `Cart` class in `app/models/cart.rb`:

```
class Cart < ActiveRecord::Base
  has_many :cart_items
  has_many :books, :through => :cart_items

  def add(book_id)
    items = cart_items.find_all_by_book_id(book_id)
    book = Book.find(book_id)

    if items.size < 1
      ci = cart_items.create(:book_id => book_id,
                             :amount => 1,
                             :price => book.price)
    else
      ci = items.first
      ci.update_attribute(:amount, ci.amount + 1)
    end
    ci
  end
```

```
  def remove(book_id)
    ci = cart_items.find_by_book_id(book_id)

    if ci.amount > 1
      ci.update_attribute(:amount, ci.amount - 1)
    else
      CartItem.destroy(ci.id)
    end
    return ci
  end

  def total
    cart_items.inject(0) {|sum, n| n.price * n.amount + sum}
  end
end
```

The `remove` method first uses a magical `find_by_attribute_name` finder method to find the cart item in the current cart that holds a certain title. Then the method uses the `update_attribute` method to make the `amount` attribute one smaller, except if the amount was already one (or less, but that shouldn't be possible). In that case, the entire cart item is deleted from the cart.

We continue the copying and slightly modifying path in `app/controllers/cart_controller.rb`:

```
class CartController < ApplicationController
  layout "catalog"
  before_filter :initialize_cart

  def add
    @book = Book.find(params[:id])
```

```
    if request.xhr?
      @item = @cart.add(params[:id])
      flash.now[:cart_notice] = "Added <em>#{@item.book.title}</em>"
      render :action => "add_with_ajax"
    elsif request.post?
      @item = @cart.add(params[:id])
      flash[:cart_notice] = "Added <em>#{@item.book.title}</em>"
      redirect_to:controller => "catalog"
    else
      render
    end
  end

  def remove
    @book = Book.find(params[:id])

    if request.xhr?
      @item = @cart.remove(params[:id])
      flash.now[:cart_notice] = "Removed 1 <em>#{@item.book.title}</em>"
      render :action => "remove_with_ajax"
    elsif request.post?
      @item = @cart.remove(params[:id])
      flash[:cart_notice] = "Removed 1 <em>#{@item.book.title}</em>"
      redirect_to :controller => "catalog"
    else
      render
    end
  end
end
```

We do the same kind of request-type sniffing here as with the `add` action. The method is almost a duplicate of the `add` action, except that this time when Ajax is used, we render the `remove_with_ajax` template, and of course, we call the `Cart#remove` method instead of `add`.

We now need to create the corresponding views, starting with `app/views/cart/remove_with_ajax.rjs`:

```
page.insert_html :top, "shopping_cart", :partial => "cart/cart_notice"
if @cart.books.include?(@book)
  page.replace_html "cart_item_#{@book.id}", :partial => "cart/item"
  page.visual_effect :highlight, "cart_item_#{@book.id}", :duration => 3
else
  page.visual_effect :fade, "cart_item_#{@book.id}", :duration => 1.5
end
page.replace_html "cart_total", "<strong>Total: $#{@cart.total}</strong>"
page.visual_effect :fade, 'cart_notice', :duration => 3
```

This time, the view is a bit more involved. We don't replace the whole shopping cart with one updated partial, but instead modify its individual objects, as follows:

- We add the notice element to the top of the cart.
- If there are still items representing the book from which we just removed one item, we update that element to show the correct amount and highlight the element. If the item was the last one of the given book, we instead fade out and finally remove the whole list item from the cart.
- We update the subtotal to match the current state of the cart.
- Finally, we slowly fade out the notice that we added in the beginning of the template.

Next, we create the view for non-Ajax operation, `cart/remove.rhtml`, which will be used as the confirmation page of the `remove` action, just as we did with `add`.

```
<strong>Please confirm removing one <em><%= @book.title %></em>
from your shopping cart.</strong>
<%= button_to "Confirm", :action => "remove", :id => params[:id] %>
```

We already created a helper for the remove link, so all we need to do to enable the functionality is to call that helper. We do that in the `cart/_item.rhtml` partial that we're using to show every item in the shopping cart:

```
<%= link_to item.book.title, :action => "show",
      :controller => "catalog", :id => item.book.id %>
<%= pluralize(item.amount, "pc", "pcs") %>,
$<%= item.price * item.amount %>

(<%= remove_book_link("-", item.book) %>)
```

Figure 5-3 shows how the removed element is faded when the remove link next to it is clicked.

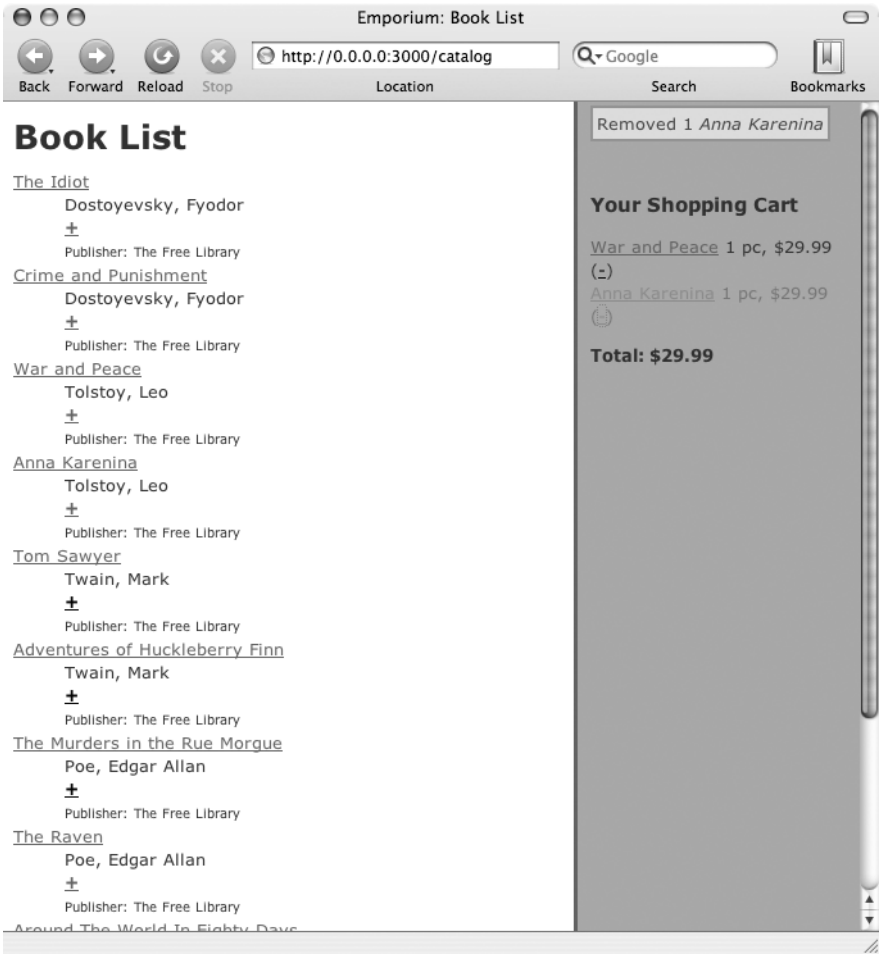


Figure 5-3. *Removing an item from the cart*

Implementing the Clear the Cart User Story

Clearing the cart follows pretty much the same path as the two previous user stories. We first add two functional tests to `test/functional/cart_controller_test.rb`, to test clearing the cart with both a normal `POST` request and an `XmlHttpRequest`.

```
def test_clearing
    post :add, :id => 4
    assert_equal [Book.find(4)], Cart.find(@request.session[:cart_id]).books

    post :clear
    assert_response :redirect
    assert_redirected_to :controller => "catalog"
    assert_equal [], Cart.find(@request.session[:cart_id]).books
  end

  def test_clearing_with_xhr
    post :add, :id => 4
    assert_equal [Book.find(4)], Cart.find(@request.session[:cart_id]).books

    xhr :post, :clear
    assert_response :success
    assert_equal 0, Cart.find(@request.session[:cart_id]).cart_items.size
  end
```

Then we add a new action called `clear` to `app/controllers/cart_controller.rb`, following along the lines of the `add` and `remove` actions.

```
def clear
  if request.xhr?
    @cart.cart_items.destroy_all
    flash.now[:cart_notice] = "Cleared the cart"
    render :action => "clear_with_ajax"
  elsif request.post?
    @cart.cart_items.destroy_all
    flash[:cart_notice] = "Cleared the cart"
    redirect_to :controller => "catalog"
  else
    render
  end
end
```

Next, we add the view for the non-Ajax way of clearing the cart, `cart/clear.rhtml`.

```
<strong>Please confirm clearing your shopping cart.</strong>
<%= button_to "Confirm", :action => "clear" %>
```

Then we move on to implement the `cart/clear_with_ajax.rjs` template.

```
page.replace_html "shopping_cart", :partial => "cart/cart"
page.visual_effect :fade, 'cart_notice', :duration => 3
```

All that the template does is to replace the whole shopping cart with a new, empty one, rendered by the `cart/_cart.rhtml` partial. Note that since we're re-rendering the whole `shopping_cart` element with a partial, we don't need to explicitly show the `cart_notification` division (as we did with the `remove_with_ajax` template), because it's already part of the `_cart.rhtml` partial template.

We'll also add a link for clearing the cart to the cart partial, but only if the cart is not empty. Remember that we created the `clear_cart_link` helper earlier in this chapter (see Listing 5-3).

```
<% if flash[:cart_notice] %>
  <%= render :partial => "cart/cart_notice" %>
<% end %>

  <h3>Your Shopping Cart</h3>

  <ul>
    <% for item in @cart.cart_items %>
    <li id="cart_item_<%= item.book.id %>">
      <%= render :partial => "cart/item", :object => item %>
    </li>
    <% end %>
  </ul>
  <p id="cart_total"><strong>Total: $<%= @cart.total %></strong></p>
  <% unless @cart.cart_items.empty? %>
  <p id="clear_cart_link">
    <small>
    <%= clear_cart_link %>
    </small>
  </p>
  <% end %>
```

Our application can now be used to add items to the shopping cart, remove them from it, and clear the whole cart with a single click.

The next natural step would be to implement the functionality of checking out—that is, finalizing the order. However, since the checkout is such a beast with credit card processing and all, we'll postpone that for now and give you all the gory details in Chapter 9.

Summary

In this chapter, we implemented a shopping cart for an online store. We used the Ajax capabilities in Ruby on Rails to implement a fast and interactive, but also backward-compatible shopping cart.

In the course of implementing the shopping cart, we showed you how to use the `has_many :through` association join models with ActiveRecord classes and how to use the Rails controller filters and store information in the `session` hash. You saw how to put Rails helpers like `pluralize` to use and how to write your own link helpers.

Our Ajax implementation demonstrated using the script.aculo.us JavaScript library, which is bundled with Rails, to create modern, Ajax-driven shopping carts. You also saw how to make sure your Ajax-driven site is also accessible to users whose browsers don't support JavaScript. Also, we covered using the Rails .rjs templates to update multiple items inside a web page with a single Ajax call.

CHAPTER 6

Forum Implementation

In this chapter, we will show you how to implement a forum that can be used by Emporium's customers to discuss book-related topics and provide feedback to George. Gathering feedback from customers is an important part of an e-commerce site, as it allows you to adapt and improve your service. For example, before implementing a new feature, you could post a message to the forum, asking your customers if they would use the feature. If the answer is no, you will save both money and time. Other uses for a forum include allowing customers to post bug reports and review books, for example.

Implementing a forum can be time-consuming. To save us some time, and save George some money, we will use the `acts_as_threaded` plugin as the basis for the forum implementation. While implementing the forum, we will also show you how to use view helpers. View helpers are a built-in feature of Rails that allow you to keep your views clean from excessive Ruby code. As usual, we will use TDD while implementing the forum, to ensure that the forum is properly tested and working according to the requirements. And as in previous chapters, we will use integration testing. In this case, we will simulate multiple users accessing the forum and posting to the forum at the same time.

Getting the Forum Requirements

We start the new day in our humble cubicle, by arranging a meeting with George to discuss the requirements for the feedback forum. George tells us that he is excited about the new possibilities the forum will offer. He yells, "It has to be simple to use. I must also be able to use the forum!" We show him a sketch of the user interface as we envision it. The create post page has three fields: name, subject, and body. George jokes, "Looks like I will be able to use that." We tell him this is as simple as it gets, and that each discussion in the forum (also referred to as a *thread*) starts from a root post. Replies to the root post are shown in a threaded fashion, so that it is easy to follow the discussion. We write down the first user stories, View Forum and Post to Forum.

Next, we show George a sketch of the View Post user story. He says, "You guys are professionals. I haven't seen the forum yet, but I can already feel that it will be something special!" We assure him that the forum will be simple and easy to use, and then continue by writing down the View Post user story.

George informs us that we have to end the meeting early. He has just received news from Jill (Emporium's best customer) that the Emporium website is down again. He has to make an emergency call to his IT department, which happens to be his nephew. Before he leaves, we show him a sketch of the Reply to Post user story. He again praises our fine sketches and runs

off towards the basement, mumbling, "I really hope that darn rat hasn't put its head through the server's processor fan again. It would be the third time this month." We try not to pay any attention to what he just said and write down the Reply to Post user story.

With George in the basement, we can start the implementation of the forum. Here are the user stories for this sprint:

- *View forum*: Users should be able to list all posts in the forum. Posts belonging to the same discussion should be shown in a threaded fashion, and the list should be sorted, with the most recent posts shown first.
- *Post to forum*: Users should be able to start a new discussion in the forum by entering their name, a subject, and the body text of the post.
- *View post*: Users that are viewing the forum should be able to click on a post and view the details.
- *Reply to post*: A customer views a forum post and decides to reply to the post by clicking the reply link. The user enters his name, the subject, and reply message, and then clicks the reply button.

We'll get started by installing the `acts_as_threaded` plugin.

Using the Threaded Forum Plugin

The `acts_as_threaded` plugin was originally developed by Bob Silva and allows you to easily implement a forum. As you've seen in previous chapters, plugins are used by developers to extend the core functionality provided by the Ruby on Rails framework. Implementing a forum with the `acts_as_threaded` plugin is as simple as most tasks in Rails. Install the plugin, and then add a database table, model, controller, and view.

The complex logic for storing forum data in a relational database, like MySQL, is handled by the plugin, allowing you to concentrate on implementing the forum, instead of writing infrastructure code. By not reinventing the wheel, you can implement the forum faster and probably with fewer bugs than if you implemented it yourself, which means a happier client in the end.

Download the plugin installation package from `www.railtie.net/plugins/acts_as_threaded.zip`. Then extract the package to the `vendor/plugins` directory, which, in our case, is `/home/george/projects/emporium/vendor/plugins`. Lastly, verify that you have the following files in the `vendor/plugins/acts_as_threaded` directory:

- `vendor/plugins/acts_as_threaded`
- `vendor/plugins/acts_as_threaded/init.rb`
- `vendor/plugins/acts_as_threaded/lib/threaded.rb`

The `init.rb` file is called by Rails on startup and initializes the plugin. The `threaded.rb` file contains the actual plugin code, which we will use to implement the forum.

Activating the plugin is as simple as adding a call to the `acts_as_threaded` method to your model, as in this example:

```
class Post < ActiveRecord::Base
  acts_as_threaded
end
```

The `acts_as_threaded` plugin adds the instance methods listed in Table 6-1 to the model.

Table 6-1. *Instance Methods Added by the* `acts_as_threaded` *Plugin*

Method	Description
`post.root?`	Returns true if the post is the root post in the thread
`post.child?`	Returns true if the post is the child of another post
`post.add_child(child)`	Adds a reply to the post
`post.children_count`	Returns the number of replies under this post
`post.full_set`	Returns an array containing the post itself and all replies under it
`post.all_children`	Returns an array containing all replies
`post.direct_children`	Returns an array containing only replies to this post

We can hear George swearing loudly in the basement, "Damn sewer rat, you never learn! Do you? This is your last... ." Then we hear a gun of some sort being fired six times. "Was that a Smith & Wesson?" we ask ourselves, before continuing with the forum implementation.

Setting Up the Forum

We have been using ActiveRecord migrations since we introduced them in Chapter 2. Continuing to use them for modifying the database schema to implement the Emporium forum is a good idea.

Updating the Database Schema

The forum functionality requires that we add a new table to the database schema, which we name `forum_posts`. This table is where all forum posts will be stored, including the information about how posts are related to each other and how they should be displayed in a hierarchical fashion.

The `forum_posts` table consists of ten columns, as shown in Figure 6-1. Table 6-2 provides a brief description of each column. You can use different column names and add as many columns as you need for storing your data, but we have followed the default structure.

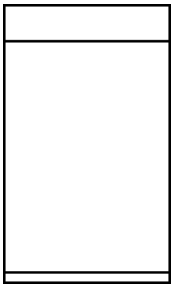


Figure 6-1. *Forum database table*

Table 6-2. *Columns in the forum_posts Table*

Column Name	Description
`root_id`	The unique identifier for the root post
`parent_id`	The unique identifier for the parent post
`lft`	The left boundary of the post; replies to this post have a left boundary that is greater than this number
`rgt`	The right boundary of the post; replies to this post have a right boundary that is less than this number
`depth`	The depth of the thread
`name`	The name of the person who created the post
`subject`	The subject of the post
`body`	The post text
`created_at`	Automatically set to the date and time the record was created
`updated_at`	Automatically set to the date and time the record was updated

■ **Tip** For in-depth information about how hierarchical data, like forum posts, can be stored in a MySQL database, see the "Managing Hierarchical Data in MySQL" article at `http://dev.mysql.com/tech-resources/articles/hierarchical-data.html`. This article also explains how the `lft`, `rgt`, and `depth` columns are used.

The first five columns—`root_id`, `parent_id`, `lft`, `rgt`, and `depth`—are used by the plugin to maintain the hierarchy of related posts. We have added the `name`, `subject`, and `body` columns, which are used to store the data entered by the user.

Recall that the `created_at` and `updated_at` columns have a special meaning in ActiveRecord. If ActiveRecord finds these columns in a table, it will automatically set the column values to the current date and time, at creation time or when the record is updated.

■ **Note** You can also name the `created_at` and `updated_at` columns `created_on` and `updated_on`; they will be treated in the same way.

Next, create the `ForumPost` model, a migration, a fixture, and a unit test by executing the `script/generate` command:

```
$ script/generate model ForumPost
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/forum_post.rb
      create  test/unit/forum_post_test.rb
      create  test/fixtures/forum_posts.yml
      exists  db/migrate
      create  db/migrate/005_create_forum_posts.rb
```

Add the code shown in Listing 6-1 to the migration file: `db/migrate/005_create_forum_posts.rb`.

Listing 6-1. *ActiveRecord Migration for the Forum Table*

```
class CreateForumPosts < ActiveRecord::Migration
  def self.up
    create_table :forum_posts do |table|
      table.column :name, :string, :limit => 50, :null => false
      table.column :subject, :string, :limit => 255, :null => false
      table.column :body, :text

      table.column :root_id, :integer, :null => false, :default => 0
      table.column :parent_id, :integer, :null => false, :default => 0
      table.column :lft, :integer, :null => false, :default => 0
      table.column :rgt, :integer, :null => false, :default => 0
      table.column :depth, :integer, :null => false, :default => 0

      table.column :created_at, :timestamp, :null => false
      table.column :updated_at, :timestamp, :null => false
    end
  end

  def self.down
    drop_table :forum_posts
  end
end
```

When the migration script is executed, it creates the `forum_posts` database table and the columns shown in Figure 6-1. Remember that you should always roll back all changes done by the migration in the `down` method. In this case, we simply delete the `forum_posts` table.

Next, perform the database migration by executing `rake db:migrate`.

```
$ rake db:migrate
```

```
(in /home/george/projects/emporium)
== CreateForumPosts: migrating ===============================================
-- create_table(:forum_posts)
   -> 0.1430s
== CreateForumPosts: migrated (0.1432s) ======================================
```

EXTENDING THE FORUM

If you have different requirements for your forum, you can add as many columns to the `forum_posts` table as you need. For example, you might add `email`, `first_name`, and `last_name` columns.

It is also easy to split the forum into separate categories, such as Feedback, News, FAQ, and Help. To set up categories, add a `category_id` column to the `forum_posts` table, which references a forum category that is stored in the `categories` table. Then add the following code to the model:

```
belongs_to :category

def self.find_all_in_category(category)
  category = Category.find_by_name(category)
  self.find :all, :conditions => "category_id =  #{category.id}"
end
```

The `belongs_to` mapping allows you to access the category the post belongs to; for example, `post.category.name`. The `find_all_in_category` method could be used to retrieve posts from a specific category; for example, `ForumPost.find_all_in_category('FAQ')`.

Modifying the Model

The `ForumPost` model created by the `generate` script is almost complete. We just need to activate the plugin and add some basic validations to the model to prevent bad data from being stored in the database. We will add validations for three fields:

- The `name` field is used for storing the name of the person who created the post, and must be between 2 to 50 characters in length.
- The `subject` field is shown on the forum main page and should be descriptive but not too long, which is why it is limited to between 5 and 250 characters.
- The `body` field should be long enough for the user to write a short message, but not too long, which is why it is limited to 5000 characters.

We activate the plugin by adding the line `acts_as_threaded` to our model, and use the `validates_length_of` validation method to enforce the length of a field. Here are the additions to make to `app/models/forum_post.rb`:

```
class ForumPost < ActiveRecord::Base
  acts_as_threaded

  validates_length_of :name, :within => 2..50
  validates_length_of :subject, :within => 5..255
  validates_length_of :body, :within => 5..5000
end
```

Notice how simple Rails and the `acts_as_threaded` plugin have made it to implement a forum. The model contains only four lines of code, and we needed to create only one table. This is enough for a basic forum implementation. You can now start using the `ForumPost` model in your controllers to add, retrieve, edit, and delete forum posts.

Unit Testing the Model

It is a good idea to have proper unit tests in place, even if you're using an external plugin that has been tested thoroughly. The plugin API might change in later releases, and this would probably go unnoticed until one of your customers informs you about the problem.

We'll create a basic test that verifies that you can create a post and also create a reply to it. Remember that the `generate` script has already created an empty unit test for us. All we need to do is add the code shown in Listing 6-2 to `test/unit/forum_post_test.rb`.

Listing 6-2. *Unit Test for the ForumPost Model*

```
require File.dirname(__FILE__) + '/../test_helper'

class ForumPostTest < Test::Unit::TestCase
  fixtures :authors

  def test_create_post_and_reply
    post = ForumPost.new( :name => 'George',
      :subject => 'Subject',
      :body => 'Body text')

    assert post.save
    assert_not_nil ForumPost.find_by_name('George')

    reply = ForumPost.new(:name => 'Jill',
      :subject => 'Reply',
      :body => 'Reply body text',
      :parent_id => post.id)

    assert reply.save
    assert reply.child?

    post.reload

    assert post.root?
    assert_equal 1, post.all_children().size
    assert_equal reply, post.all_children()[0]
  end
end
```

Notice that the test uses the `authors` fixture that we created in Chapter 2. First, the unit test creates a new post, which is the root post of a new thread. Then it saves the post and verifies that it was successfully saved to the database with an `assert`. Recall that the `save` method returns `false` if the object was not saved successfully. We double-check that the post can be found with the `find_by_name` dynamic finder method (introduced in Chapter 3). The unit test also verifies that the first post is the root post by calling `root?`. This method returns `true` if the post has no parent posts.

We also want to be sure that someone can reply to the post, which is why we create a reply to the root post. This is done by creating a new post and setting the `parent_id` column to the `id` of the root post. After saving the reply post, we verify that the post really is a reply to the first post by calling the `child?` method. This method returns `true` if the post is a child, or reply, of another post.

At the end of the unit test, we verify that the root post has exactly one child post, and that the first post returned by `all_children` returns the reply.

Before running the tests, clone the development database structure to the test database by executing the following command:

```
rake db:test:clone_structure
```

Next, run the unit test as follows:

```
$ ruby test/unit/forum_post_test.rb
```

```
Loaded suite test/unit/forum_post_test
Started
.
Finished in 0.101902 seconds.

1 tests, 7 assertions, 0 failures, 0 errors
```

Now we are ready to create the controller and view.

Generating the Controller and View

George has given us a tight schedule. We could create the forum controller and view files manually, but instead, we decide to save some time and use the `generate` script. Run the `generate` script with the following parameters to generate the controller:

```
$ script/generate controller Forum index reply show post create
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/forum
      exists  test/functional/
      create  app/controllers/forum_controller.rb
      create  test/functional/forum_controller_test.rb
      create  app/helpers/forum_helper.rb
      create  app/views/forum/index.rhtml
      create  app/views/forum/reply.rhtml
      create  app/views/forum/show.rhtml
      create  app/views/forum/post.rhtml
      create  app/views/forum/create.rhtml
```

Note that each action in the controller maps to a user story:

- The `index` action implements the View Forum user story.
- The `reply` action implements the Reply to Post user story.
- The `show` action implements the View Post user story.
- The `post` and `create` actions implement the Post to Forum user story.

Open `app/controllers/forum_controller.rb` in your editor. You should see the following code (note that we have left out part of the file):

```
class ForumController < ApplicationController

  def index
  end

  def reply
  .
  .
end
```

Next, we will implement each of the forum user stories.

Implementing the User Stories

As in previous chapters, we will show you how to use the TDD approach while implementing the forum. This means that you will first create the unit tests, or in this chapter, integration tests. In Chapter 3, we introduced integration testing and mentioned some of the benefits of using integration tests, including multiple session support and tests that span multiple controllers. A forum can be accessed by many users at the same time, which makes it a good candidate for integration testing. For example, George might be replying to a post at the same time as Jill is creating a new post. Integration tests make it possible to simulate this by allowing us to open multiple sessions in the test and execute each action in a different session.

The forum needs some posts for the other user stories to make any sense, so we'll start by implementing the Post to Forum user story.

Implementing the Post to Forum User Story

The Post to Forum user story describes how a customer, or George himself, submits a new post to the forum. This is done by filling out the required information, including the name of the person creating the post, plus the subject and body of the post. To publish the post to the forum, the user should click the Post button. This saves the post in the database and redirects the user to the main page of the forum, where he can see the post at the top of the page.

Creating the Integration Test

First, create the integration test with the `generate` script:

```
$ script/generate integration_test Forum
```

```
      exists  test/integration/
      create  test/integration/forum_test.rb
```

If you open the generated test, you can see that it contains one dummy test, which should be removed.

Next, create the new testing DSL for the forum by adding the code shown in Listing 6-3 to `test/integration/forum_test.rb`.

Listing 6-3. *First Version of the Integration Test*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class ForumTest < ActionController::IntegrationTest

  def test_forum
  end

  private

  module ForumTestDSL
    attr_writer :name
  end

  def new_session_as(name)
    open_session do |session|
      session.extend(ForumTestDSL)
      session.name = name
      yield session if block_given?
    end
  end

end
```

The `new_session_as` method is used for opening a new session for a user. This can be used to simulate George starting his browser and going to the Emporium website, for example. The `test_forum` method is the main method for the integration test that will use the DSL to test all the user stories as a whole.

Next, add the `post_to_forum` method to the `ForumTestDSL` directly after the line `attr_writer :name`:

```
    def post_to_forum(parameters)
      get "/forum/post"
      assert_response :success
      assert_template "forum/post"

      post "/forum/create", parameters

      assert_response :redirect
      follow_redirect!
      assert_response :success
      assert_template "forum/index"
      return ForumPost.find_by_subject(parameters[:post][:subject])
    end
```

The post_to_forum method simulates a user creating a new post. First, it opens the URL /forum/post and verifies that it works simply by checking the HTTP status code. Then it creates a new post by calling the /forum/post URL. Next, the test verifies that the request was successful and that there is a redirect to the forum main page. At the end, the method returns the post object that was created by the test, so that we can use it later in the test.

Next, put the new method to use by changing the test_forum method as follows:

```
  def test_forum
    jill = new_session_as(:jill)
    post = jill.post_to_forum :post => {
      :name => 'Bookworm',
      :subject => 'Downtime',
      :body => 'Emporium is down again!'
    }
  end
```

This will test the Post to Forum user story. It simulates Jill creating a new post on the forum. Note that we are saving the post for later use in the integration test, so that we can reply to it.

You can now run the integration test, in true TDD style (it should fail):

```
$ ruby test/integration/forum_test.rb
```

```
Loaded suite test/integration/forum_test
Started
F
Finished in 0.061169 seconds.

  1) Failure:
test_forum(ForumTest)
    [test/integration/forum_test.rb:26:in 'post_to_forum'
     test/integration/forum_test.rb:7:in 'test_forum'
     /usr/lib/ruby/gems/1.8/gems/actionpack-1.12.1/lib/action_controller/ ➥
integration.rb:427:in `run']:
Expected response to be a <:redirect>, but was <200>

1 tests, 3 assertions, 1 failures, 0 errors
```

It fails because we haven't implemented anything but the test yet. Now let's complete the user story by modifying the controller and views that were created by the generate script.

Completing the Controller

The controller contains two actions that are used in the Post to Forum user story. One displays the form, and the other takes the user input and persists the post to the database. Both the `post` and `create` actions exist already, but they contain no code, so modify `app/controllers/forum_controller.rb` as follows:

```
def post
  @page_title = 'Post to forum'
  @post = ForumPost.new
end

def create
  @post = ForumPost.new(params[:post])
  if @post.save
    flash[:notice] = 'Post was successfully created.'
    redirect_to :action => 'index'
  else
    @page_title = 'Post to forum'
    render :action => 'post'
  end
end
```

The `post` action does one thing: it creates a new `ForumPost` object that is used by the form tags in the view. The `create` action is a bit more complex. It receives the form input from the user and creates a new `ForumPost` object. It then tries to save it to the database. If the user-supplied data passes validation, it redirects the user to the forum main page. If there are any validation errors, it renders the create post page instead, where we use the `error_messages_for` helper to show the validation errors to the user.

Creating the View

Recall that the Post to Forum user story is implemented with the controller's `post` action, which directly maps to the `app/views/forum/post.rhtml` view. The ERB code for this view is shown in Listing 6-4. Save the code in `app/views/forum/post.rhtml`.

Listing 6-4. *View for the Post to Forum User Story*

```
<%= error_messages_for 'post' %>
<%= start_form_tag :action => 'create' %>
    <%= hidden_field :post, :parent_id %>
    <p><label for="post_name">Name</label><br/>
    <%= text_field 'post', 'name'  %></p>
    <p><label for="post_subject">Subject</label><br/>
    <%= text_field 'post', 'subject'  %></p>
    <p><label for="post_body">Body</label><br/>
    <%= text_area 'post', 'body'  %></p>

  <%= submit_tag "Post" %>
<%= end_form_tag %>
```

The view uses the built-in Ruby on Rails form helpers `hidden_field`, `text_field`, and `text_area` for creating the form fields. Note that we are using label tags and that the label tag's `for` attribute is used to associate the label with a form field. Using label tags in forms is important because it improves usability, such as by extending the clickable area to outside the field. The hidden field in the view is used by the Reply to Post user story, which we will implement later in this chapter.

■ **Tip** For more information about form helpers, see the online documentation at: `http://api.rubyonrails.org/classes/ActionView/Helpers/FormHelper.html`.

Testing

Let's run the integration test and verify that our user story is functioning as specified:

```
$ ruby test/integration/forum_test.rb
```

```
Loaded suite test/integration/forum_test
Started
.
Finished in 0.114117 seconds.

1 tests, 5 assertions, 0 failures, 0 errors
```

If you followed our instructions, the test should pass without errors.

If you want, you can try a manual test as well. Manual testing is an important complement to automated tests and also makes development more fun, as you can see what you are building. Open `http://localhost:3000/forum/post` in your browser. Enter your name, the subject, and the body of the post in the form. Figure 6-2 shows our validation at work: an error message appears because we tried to post a message without entering any data into the fields.

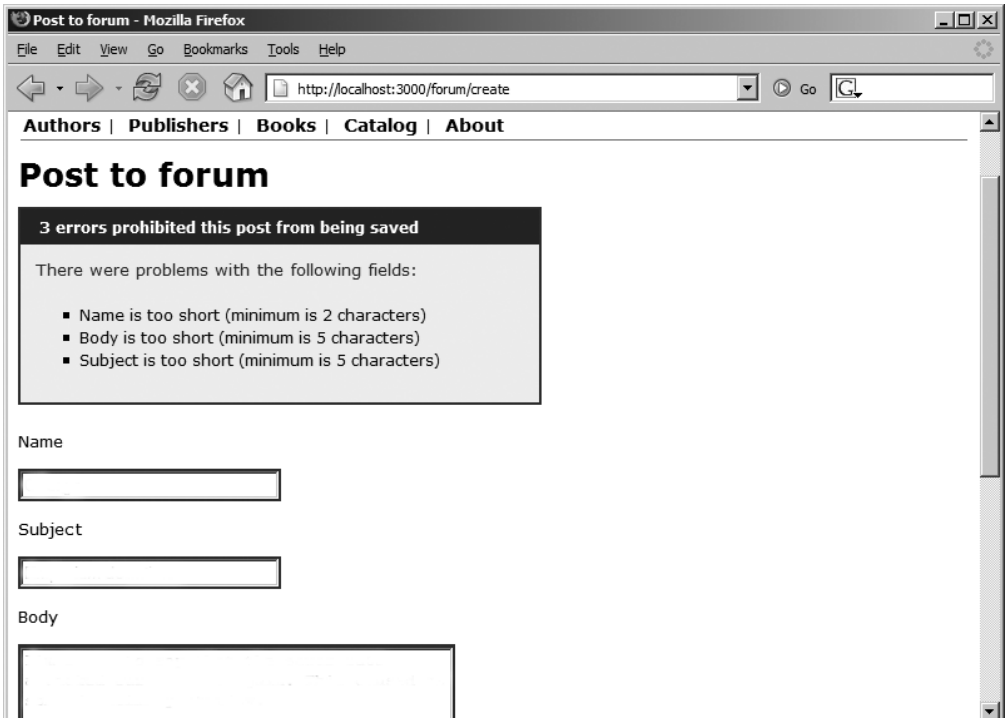


Figure 6-2. *The create post page showing some validation errors*

When you click the Post button, and if the data you entered passes validation, you're redirected to `http://localhost:3000/forum`. The page displays the message `Forum#index`, as we have not yet implemented the View Forum user story.

■ **Note** Here, we showed you how to implement a test that verifies that a valid post is created successfully. It would be a good idea to also create tests for other scenarios, such as when data fails validation.

Implementing the View Forum User Story

The View Forum user story describes how a user can access the forum's main page to view a list of the most recent posts. The page displays all the forum posts in a threaded fashion, with the most recent post shown at the top of the list, similar to this example:

```
Hello from the Cayman Islands by George * 02/02/2006
-->Bring a keg of Rum by Jill * 02/03/2006
---->Is one enough? by George * 02/05/2006
WARNING! Rat poison deployed in server room by George * 31/12/2005
```

George has great plans for Emporium. He envisions a lot of traffic and users posting exotic questions on the forum, so showing all posts on the same page is not very wise. This is easy to fix with pagination, which we will use to ensure that no more than 20 posts are shown at the same time.

Updating the Integration Test

As explained earlier, integration tests are great for testing multiple sessions. The test that we'll create next will check this by simulating both George and Jill accessing the forum at the same time. Add the following code to the DSL in `test/integration/forum_test.rb`, immediately after the `post_to_forum` method:

```
    def view_forum
      get "/forum"

      assert_response :success
      assert_template "forum/index"
      assert_tag :tag => 'h1', :content => 'Forum'
      assert_tag :tag => 'a', :content => 'New post'
    end
```

This method simply accesses the forum main page and verifies that the heading on the page is displayed correctly. The main forum page should also contain a link to the create post page. This is verified with another `assert_tag` that looks for a link named `New post`. Change the `test_forum` method as follows (changes are highlighted):

```
  def test_forum
    jill = new_session_as(:jill)
    george = new_session_as(:george)
    post = jill.post_to_forum :post => {
      :name => 'Bookworm',
      :subject => 'Downtime',
      :body => 'Emporium is down again!'
    }
    george.view_forum
    jill.view_forum
  end
```

This simulates George opening the forum in his browser right after Jill has posted to the forum. In later tests, we'll create a test that simulates George replying to the post made by Jill.

Next, run the test by executing `ruby test/integration/forum_test.rb`. You should see the test fail with the following message:

```
expected tag, but no tag found matching {:content=>"New post", :tag=>"a"} in:
```

The test is expecting to find the link New post, but fails because we haven't modified the view. Note that the test didn't fail on the first `assert` because the heading is `Forum#index`.

Modifying the View

Fix the view by opening `app/views/forum/index.rhtml` and changing the contents of the file to match Listing 6-5.

Listing 6-5. *The View for the View Forum User Story*

```
<% if @posts.size > 0 %>
<div><%= link_to 'New post' , :action => 'post' %></div>
<p>
<%= display_as_threads @posts %>
</p>
<% else %>
There are no posts yet. <%= link_to 'Be the first one to post here' , ➥
:action => 'post' %>
<% end %>
<br/>

<%= link_to 'Previous page' , { :page => @post_pages.current.previous } ➥
if @post_pages.current.previous %>
<%= link_to 'Next page' , { :page => @post_pages.current.next } ➥
if @post_pages.current.next %>
```

Run the integration test again with the following command: `$ ruby test/integration/forum_test.rb`. This time, you should see the test fail with the following message:

```
Expected response to be a <:success>, but was <500>
```

To find the cause for this, open your browser and go to `http://localhost:3000/forum`. As shown in Figure 6-3, the page shows a nice and detailed error message of where the error is located.

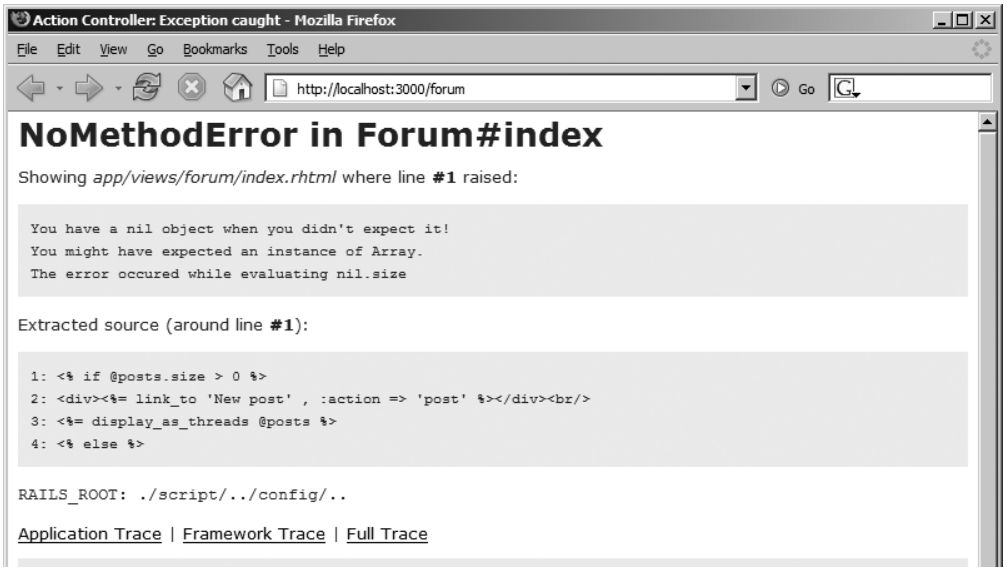


Figure 6-3. *Rails error page*

■ **Tip** The error page that is displayed when exceptions are thrown is exceptionally well thought out. It shows you details about the request, response, and session. At the bottom of the page, you can also see the stack trace leading up to the error, including links that allow you to filter the stack trace to show only your application code, the Rails framework's code, or the full trace.

The error page tells you that line 1 contains an error and shows you the code around line 1 in your browser. As you can see, the page generates an error when it tries to call the `size` method on the `@posts` instance variable. This is because we haven't initialized the `@posts` variable in the controller. Fixing the page requires completing the following tasks:

- Change the `index` action in the controller to retrieve a paginated list of posts.
- Create a view helper containing the `display_as_threads` method. This method is used to display the forum posts.

Modifying the Controller

Start by changing the `index` action. Open `app/controllers/forum_controller.rb` in an editor and modify the `index` action as follows:

```
  def index
    @page_title = 'Forum'
    @post_pages, @posts = paginate :forum_posts, :per_page => 20, ➥
 :order => 'root_id desc, lft'
  end
```

The `paginate` method is used to retrieve a paginated list of `ForumPost` objects. Furthermore, we specify that we want to show a maximum of 20 posts per page, and that the posts should be ordered by the `root_id` and the `lft` columns.

You can try to access the page again in your browser or run the integration test. Both should fail with an error being shown, because the `display_as_threads` method is not found.

Using a View Helper

Displaying a list of posts requires a fair amount of code, as each post can have one or more replies, and these need to be shown as a thread. To avoid cluttering the view with too much Ruby code, we will use a view helper to generate the list of posts.

■ **Tip** Putting too much code in the view is usually a bad idea. If you see that your view is getting cluttered with Ruby code, you can refactor it and move the code to a view helper, controller, or another part of the system.

The `generate` script already created a view helper for you. Open `app/helpers/forum_helper.rb` and add the `display_as_threads` method, as follows:

```
module ForumHelper
  def display_as_threads(posts)
    content = ''
    for post in posts
      url = link_to("#{h post.subject}", {:action => 'show', :id => post.id})
      margin_left = post.depth*20
      content << %(
      <div style="margin-left:#{margin_left}px">
        #{url} by #{h post.name} &middot; ➥
#{post.created_at.strftime("%H:%M:%S %Y-%M-%d") }
      </div>)
    end
    content
  end
end
```

The new `display_as_threads` method takes an array of `post` objects as a parameter. Recall that we are calling this method from the view and passing the `posts` variable to it, which is initialized by the controller. The method loops through all the posts and creates the HTML necessary for displaying the posts in a threaded fashion.

Note that each post is located inside a `div` tag that is indented to the right by specifying a left margin. This is done with the `margin-left` CSS property. The size of the indentation is related to how deep down the reply is in the hierarchy; each reply is indented 20 pixels to the right of the parent. Also note that the `link_to` method is used to create a link to the show post page, which we will implement in the next section.

Open your browser and create a couple of posts. You should see a page similar to the one shown in Figure 6-4. Recall that in Chapter 2, we changed the layout to show the flash message at the top of the page.

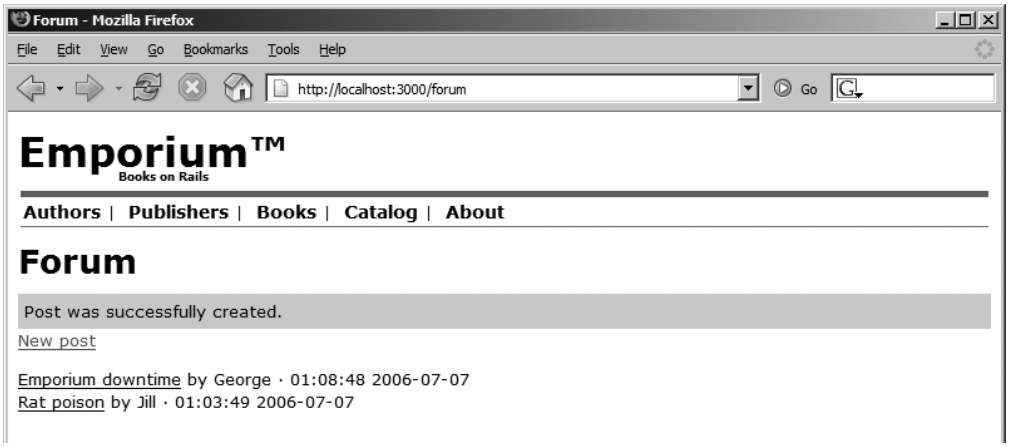


Figure 6-4. *The forum main page*

You can now run the test again by executing `ruby test/integration/forum_test.rb`. You should see the test pass without errors or failures.

```
$ ruby test/integration/forum_test.rb
```

```
Loaded suite test/integration/forum_test
Started
.
Finished in 0.183961 seconds.

1 tests, 13 assertions, 0 failures, 0 errors
```

You can now both create and view a list of posts. The next logical thing to implement is the View Post user story.

Implementing the View Post User Story

On the forum main page, you can see the posts you have created. If you click the post subject, you are taken to a page that displays the text `Forum#show`. This page was created by the `generate` script and is clearly not what we want. We need to modify the page so that it shows all the information about the post.

Updating the Integration Test

Start by modifying the integration test. As we said earlier, most forums have more than one user. We will therefore continue simulating a scenario where Jill is creating a post and George is browsing to the post details page. Add the new `view_post` method to `test/integration/forum_test.rb`.

```
    def view_post(post)
      get "/forum/show/#{post.id}"

      assert_response :success
      assert_template "forum/show"
      assert_tag 'h1', :content => "'#{post.subject}'"
    end
```

The new DSL method takes a `post` object as a parameter, which it uses to access the post details page. On the post details page, the test verifies that the subject of the post can be found in the page content.

Next, we want to simulate George browsing the forum and viewing the post Jill created. Add a call to the `view_post` action, as follows (highlighted):

```
  def test_forum
    jill = new_session_as(:jill)
    george = new_session_as(:george)
    post = jill.post_to_forum :post => {
      :name => 'Bookworm',
      :subject => 'Downtime',
      :body => 'Emporium is down again!'
    }
    george.view_forum
    jill.view_forum

    george.view_post post
  end
```

Running the test now would cause the test to fail miserably with the following error:

```
expected tag, but no tag found matching {:tag=>"h1", :content=>"'Downtime'"} in:
```

This is because you have not yet modified the `show` action and view to show the details of the post. Remember that we created these earlier with the help of the `generate` script.

Modifying the Controller

Modify the `show` action in `app/controllers/forum_controller.rb` so that it loads the specified forum post. This is done with a call to `ForumPost.find`:

```
  def show
    @post = ForumPost.find(params[:id])
    @page_title = "'#{@post.subject}'"
  end
```

As usual, we set the page title using the `@page_title` variable.

Modifying the View

Change the contents of the `app/views/forum/show.rhtml` to be as follows:

```
<dl>
  <dt>Name</dt>
  <dd><%= h @post.name %></dd>
  <dt>Subject</dt>
  <dd><%= h @post.subject %></dd>
  <dt>Body</dt>
  <dd><%= h @post.body %></dd>
</dl>
<%= link_to 'Reply', :action => 'reply', :id => @post %> |
<%= link_to 'Back', :action => 'list' %>
```

■ **Note** It is extremely important that you escape all data that is entered by users with the `h` method. This helps in protecting you from security risks, such as cross-site scripting attacks. For more information about how to protect your site against various attacks, refer to Chapter 8, "Security".

With the action and view in place, the integration test should now work, so run it once more to verify that the integration test passes:

```
$ ruby test/integration/forum_test.rb
```

```
Loaded suite test/integration/forum_test
Started
.
Finished in 0.145623 seconds.

1 tests, 11 assertions, 0 failures, 0 errors
```

You should also perform a manual test to verify that the page works correctly. Open `http://localhost:3000/forum/show/1` (you might have to change the `id`), and you should see the details of the post, as shown in Figure 6-5. Notice that there is a Reply link at the bottom of the page.

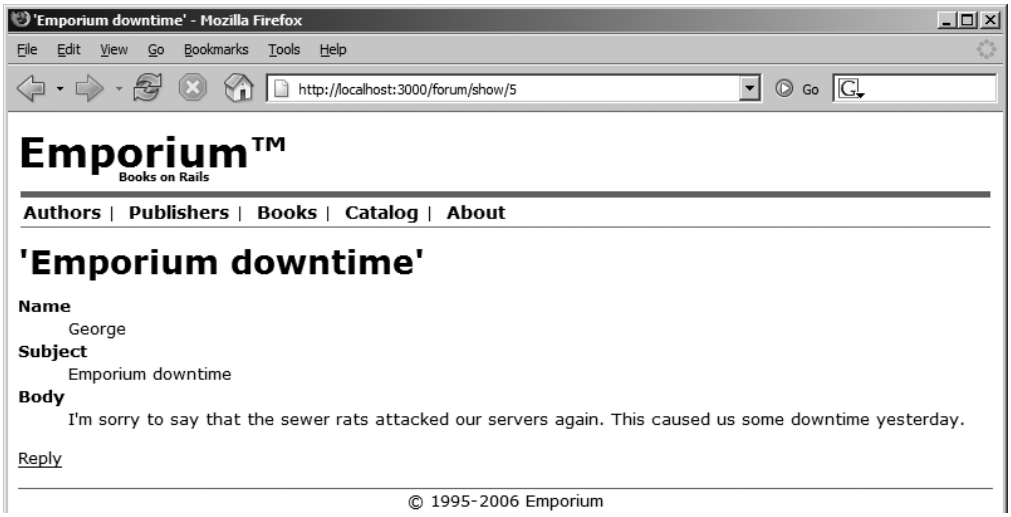


Figure 6-5. *The final version of the post page*

The Reply to Post user story is the last user story that we need to implement.

In case you want to display links to replies (to the current post) on the Show Post page, you can add the following code to the view:

```
<%= display_as_threads @post.direct_children%>
```

This displays the replies to the current post, but only the direct replies. To display all replies use the following code:

```
<%= display_as_threads @post.all_children%>
```

Implementing the Reply to Post User Story

After, for example, Jill has posted a message to the forum, George should be able to view the post and reply to it. George should be able to do this by clicking the Reply link on the post details page (see Figure 6-5).

The Reply to Post user story requires that you change the integration test one last time. Add the `reply_to_post` method to the DSL:

```
def reply_to_post(post, parameters)
  get "/forum/reply/#{post.id}"
  assert_response :success
  assert_tag 'h1', :content => "Reply to '#{post.subject}'"

  post "/forum/create/#{post.id}", parameters

  assert_response :redirect
  follow_redirect!
  assert_response :success
  assert_template 'forum/index'
  assert_tag :a, :content => post.subject
end
```

The `reply_to_post` method simulates a user browsing to the reply to post page. The test verifies that the page title is correct and checks that there is a redirection to the forum main page, which indicates that the post was saved successfully. At the end, the test verifies that the main page lists the post that was created, by looking for an anchor tag having the same content as the post's subject.

Next, change the `test_forum` method as follows:

```
def test_forum
  jill = new_session_as(:jill)
  george = new_session_as(:george)
  post = jill.post_to_forum :post => {
    :name => 'Bookworm',
    :subject => 'Downtime',
    :body => 'Emporium is down again!'
  }
  george.view_forum
  jill.view_forum

  george.view_post post
  george.reply_to_post(post, :post => {
      :name => 'George',
      :subject => 'Rats!',
      :body => 'Rats!!!!!!!!'
    }
  )
end
```

This will create a reply to the post Jill made earlier in the test. But when the integration test is run, it should fail with the following error message:

```
expected tag, but no tag found matching {:content=>"Reply to 'Downtime'", :tag=>"h1"
```

The test is failing because the `reply_to` action is empty, when it should be creating a new post and setting the correct page title. Fix this by modifying the `reply` action:

```
def reply
  reply_to = ForumPost.find(params[:id])
  @page_title = "Reply to '#{reply_to.subject}'"
  @post = ForumPost.new(:parent_id => reply_to.id)
  render :action => 'post'
end
```

First, the `reply_to` action loads the `post` object that the user is replying to from the database. This is because we want the page title to be the subject of the post we are replying to, which is done by setting the `page_title` instance variable. Then we create a new `post` object, which is the reply, and set the `parent_id` to be the `id` of the post to which we are replying.

Execute the test one final time, and you should see it pass without any errors.

As usual, do a quick manual test by replying to one of the posts you have already created. This opens a URL pointing to `http://localhost:3000/forum/reply/1` (note that your `id` parameter might be different) in your browser, as shown in Figure 6-6.

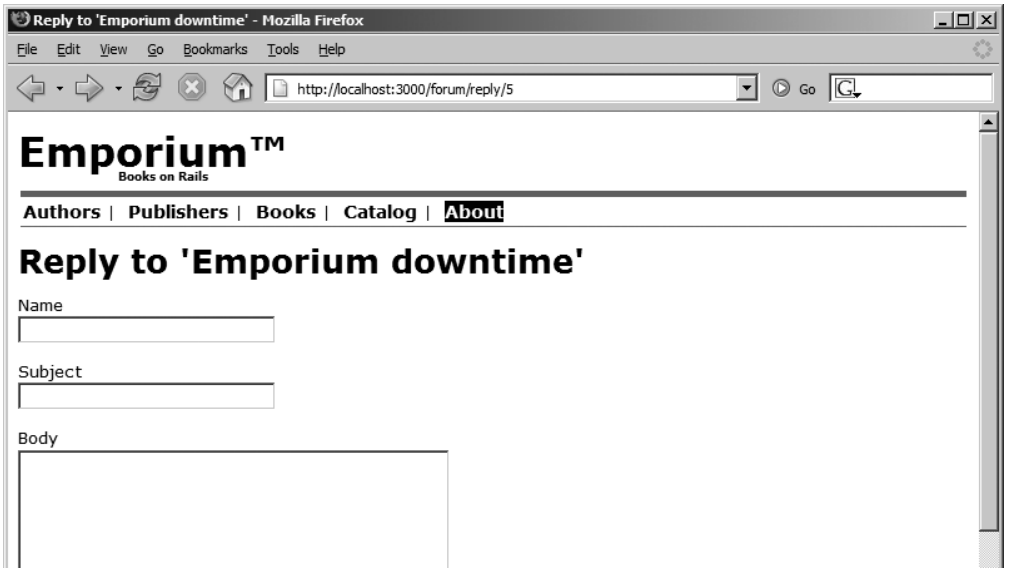


Figure 6-6. *The reply to post page*

After you reply to the post, you should be redirected to the main page, where you should see something similar to Figure 6-7.

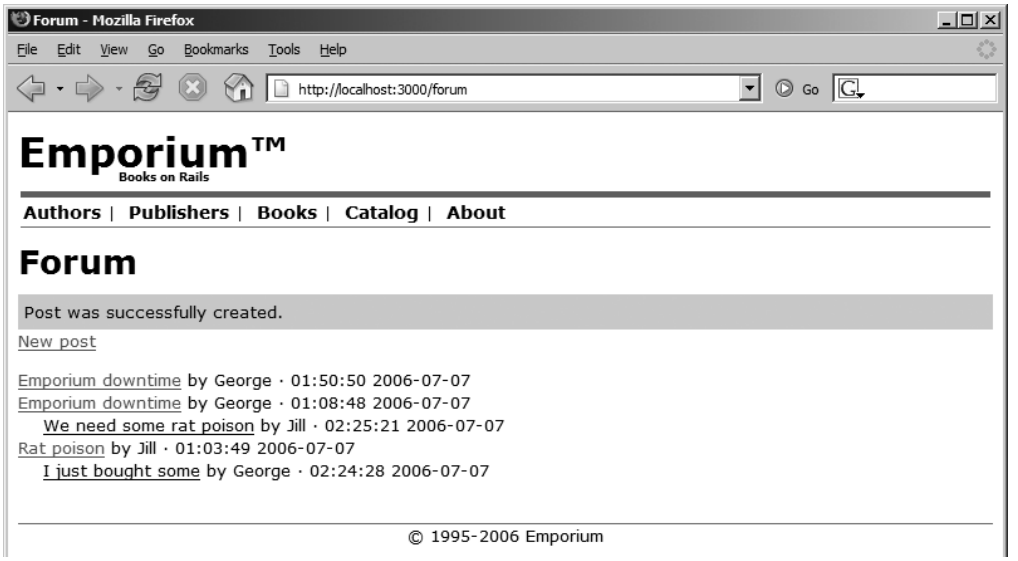


Figure 6-7. *The forum main page showing a discussion thread*

We notice that George has come up from the basement and that he is holding a Smith & Wesson, just as we suspected. He yells, "There's nothing that beats a Smith & Wesson—not even a rabid sewer rat!"

We ask him to do an acceptance test of the forum functionality. He seems happy with how easy it is to use, and tells us that we have done a good job.

Summary

In this chapter, we showed you how to implement a forum with the help of a third-party plugin. We also demonstrated how to create integration tests that simulate multiple users accessing the forum and how to create a view helper to display the posts in the forum as threads.

In the next chapter, you will learn how to implement tagging.

CHAPTER 7

Tagging Support

In this chapter, we will extend the Emporium site to support tagging. Tagging, which is done by assigning a set of tags (keywords) to an entity, is used by some of the most respected websites on the Internet. For example, Amazon, Yahoo, and Google allow their users to categorize and link together a variety of information with the help of tags.

Adding tagging functionality to an e-commerce site has several benefits. One benefit is that tagging simplifies the categorization of content. A good showcase for the benefits of tagging is del.icio.us, which uses a collaborative form of tagging (folksonomy) where all users are allowed to bookmark websites and assign tags to them. The success of del.icio.us can largely be attributed to tagging, because it simplified the way people could categorize and find their bookmarks.

Another benefit of tagging is that it allows you to make recommendations to your customers. Say, for example, that your customer is browsing a book on programming that has been tagged with the keywords `Ruby` and `Programming`. You can then show other books that share these tags, and this will probably increase your sales.

Getting the Tagging Requirements

Back when we were getting George's requirements for the book inventory management system (in Chapter 3), George told us that he wanted the online store to recommend related items when users look for books, as he has seen on Amazon. Now we're ready to tackle the implementation of this feature in this sprint.

George tells us "I feel really stupid today! I was going to buy just one book on home brewing from Amazon, but they tricked me into ordering both *The Brewmaster's Bible: Gold Standard for Home Brewers* and *Homebrewing for Dummies*. I was viewing the details of *The Brewmaster's Bible*, when I noticed that they recommended *Homebrewing for Dummies*, and I just had to buy it!" He hopes that adding this feature will have the same effect on his customers and that it will double his sales.

We agree with him that this is a good idea, and tell him that this would be easy to implement with a tagging system. We also point out that it can be used for categorizing books, so that customers can browse a list of categories. After a brief discussion, we come up with the following user stories:

- *Assign tags*: George must be able to assign a set of tags to a book by typing in a comma-delimited list of tags on the add book page. This extends the Add Book user story we implemented in Chapter 3.
- *Edit tags*: George must be able to remove or add tags to an existing book on the edit book page. This should be possible by editing the Tags text field. This extends the Edit Book user story we implemented in Chapter 3.
- *List tags*: Emporium's customers must be able to view all tags that have been used in the system.
- *Show tag*: While viewing the details of a book, a customer must be able to click each tag and display books having the same tags.
- *Recommend books*: While the user is viewing a book, the system must be able to recommend similar books to the user. The system must provide links to both books and tags that are related to the current book.

We tell George that it will take about a day to implement the tagging functionality. George responds, "No worries. It can cost a million dollars, as long as it doubles my sales."

Using the Tagging RubyGem

Implementing a tagging system is a complex task and requires a fair amount of code and SQL to be written. Fortunately, we can save days (or weeks) of coding and bug fixing by using the `acts_as_taggable` gem. `acts_as_taggable` is an ActiveRecord mix-in that allows you to add tagging capabilities to your ActiveRecord models. Originally coded by Demetrius Nunes, `acts_as_taggable` is an open source project hosted by RubyForge. The API documentation can be found at `http://taggable.rubyforge.org`, and the project's homepage is `http://rubyforge.org/projects/taggable/`.

■ **Note** Don't confuse the `acts_as_taggable` gem (`http://rubyforge.org/projects/taggable/`) with the `acts_as_taggable` plugin (`http://dev.rubyonrails.com/svn/rails/plugins/acts_as_taggable/`). Both the gem and plugin have similar features. We chose to use the gem because, at the time of writing, the plugin was lacking some of the more advanced features that we showcase in this book.

To install `acts_as_taggable`, simply execute the following command:

```
$ sudo gem install acts_as_taggable
```

```
Bulk updating Gem source index for: http://gems.rubyforge.org
Successfully installed acts_as_taggable-2.0.2
Installing ri documentation for acts_as_taggable-2.0.2...
Installing RDoc documentation for acts_as_taggable-2.0.2...
```

The location of the repository depends on your system and RubyGems configuration. On our machine, the gem was installed in `/usr/lib/ruby/gems/1.8/gems/acts_as_taggable-2.0.2/`.

■ **Tip** If desired, you can turn the gem into a plugin. First, change the current directory to the `vendor/plugins` directory with `cd vendor/plugins`, and then execute `gem unpack acts_as_taggable`. Next, create a file called `init.rb` in the `acts_as_taggable` folder and put the following code in it: `require 'taggable'`.

We need to tell Rails to load the `acts_as_taggable` gem at startup, since we are using a gem and not a Rails plugin. To do this, add the line `require_gem 'acts_as_taggable'` to the last line of `config/environment.rb`, as shown here:

```
# Include your application configuration below
require_gem 'acts_as_taggable'
```

After you have saved the changes and restarted WEBrick, you can specify that an ActiveRecord model should be taggable by adding the line `acts_as_taggable` to the code. This gives you access to the instance methods shown in Table 7-1 and the class methods shown in Table 7-2.

■ **Tip** See the online documentation at `http://taggable.rubyforge.org/` for the complete and latest version information about `acts_as_taggable`.

Table 7-1. *Instance Methods Introduced by acts_as_taggable*

Method	Description
`book.tag(tags, options)`	Assigns the specified tags to the book. This is done by parsing the `tags` parameter. The `tags` parameter is a string, and tags are separated by spaces. A different separator can be used by adding the `:separator => ' '` option to the `options` hash. Add the `:clear => true` option to remove all existing tags before assigning the new ones.
`book.tag_names(reload)`	Returns an array of tags that have been assigned to the book. The collection can be forced to be reloaded from the database by setting `reload` to `true`.
`book.tagged_related(options)`	Finds books that share most of the same tags as the current book. The `options` hash can be used for specifying how many books should be returned by the method. The default is five. To specify a different value, add `:limit => n` to the `options` hash. Note that you need to specify the separator with the `separator` parameter, if you are not using the default separator.
`book.tagged_with?(tag_name, reload)`	Returns `true` if the book has been assigned the specified tag. The `reload` parameter can be used to force a reload of the book, before the check is performed.

■ **Tip** The RubyGems packaging system provides a handy command that can be used for accessing the documentation of packages installed with RubyGems, including `acts_as_taggable`. This is useful, for example, if you are working offline and can't read the online documentation. Execute `gem_server` to start a web server on your local machine, which allows you to access the documentation at `http://localhost:8808`.

Table 7-2. *Class Methods Introduced by acts_as_taggable*

Method	Description
`Book.find_related_tagged(book, options)`	Finds related books and returns them as an array. The method returns the five books that share the most tags with the specified book. The `options` hash can be used to set the maximum amount of books returned.
`Book.find_related_tags(tags, options)`	Returns tags that are related to the ones specified with the `tags` parameter. The `options` hash can be used to set the maximum amount of tags returned.
`Book.find_tagged_with(options)`	Finds books that are tagged with the specified options. This can be used to find books that have any or all of the specified tags, for example. See the online documentation for a complete list of available options.

Setting Up for Tagging

As in previous chapters, we will use ActiveRecord migrations to modify the database schema to implement tagging. We will add tables, create a model, and develop unit tests. We'll also introduce testing with the console.

Updating the Database Schema

When George assigns a set of tags to a book, we must be able to store them somewhere, and also be able to associate them with the book. For this purpose, we will add two new tables to the database schema (see Figure 7-1):

- The `tags` table is where the unique `id` and `name` of all tags are stored.
- The `books_tags` table is used to associate a set of tags with one or more books, through a many-to-many relationship (ActiveRecord database relationships are covered in Chapter 3). The `books_tags` table includes foreign key references to the `tags` and `books` tables.

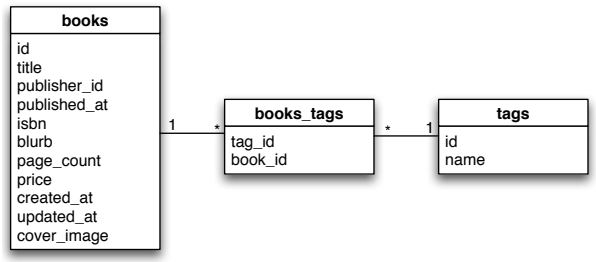


Figure 7-1. *Tables used by the tagging system*

Next, create the migration by executing the `generate` command:

```
$ script/generate migration CreateTagsAndBooksTags
```

```
    exists db/migrate
    create db/migrate/006_create_tags_and_books_tags.rb
```

The `generate` script creates an empty migration script for you. Next, change the script as shown in Listing 7-1.

Listing 7-1. *Migration Script for the Tagging Functionality*

```
class CreateTagsAndBooksTags < ActiveRecord::Migration
  def self.up
    create_table :tags do |table|
      table.column :name, :string, :limit => 255, :null => false, :unique => true
    end

    create_table :books_tags, :id => false do |table|
      table.column :tag_id, :integer, :null => false
      table.column :book_id, :integer, :null => false
    end

    say_with_time 'Adding foreign keys' do
      # Add foreign key reference to books_tags table
      execute 'ALTER TABLE books_tags ADD CONSTRAINT fk_tb_tags ➥
FOREIGN KEY ( tag_id ) REFERENCES tags( id ) ON DELETE CASCADE'
      execute 'ALTER TABLE books_tags ADD CONSTRAINT fk_tb_books ➥
FOREIGN KEY ( book_id )
REFERENCES books( id ) ON DELETE CASCADE'
    end
    say_with_time 'Adding default tags' do
      execute(insert_tags_sql)
    end
  end

  def self.down
    drop_table :books_tags
    drop_table :tags
  end

  def self.insert_tags_sql
    <<-END_OF_DATA
insert into tags values
(1,"Romance"),
(2,"Cooking"),
(3,"Mystery"),
(4,"History"),
(5,"Politics"),
(6,"Elvis"),
(7,"Science Fiction")
END_OF_DATA
  end
end
```

The script creates the two tables, `tags` and `books_tags`. The migration also adds foreign key references to the `books` and `tags` tables by executing raw SQL with the `execute` command. At the end of the migration, the script adds a default set of tags to the `tags` table by again calling the `execute` command.

■ **Note** The `create_table` method creates an `id` column by default. When creating the `books_tags` join table, we are telling the `create_table` command not to add an `id` column by setting the `id` parameter to `false`.

Now, run the migrations by executing the following command:

```
rake db:migrate
```

You should see the command run without errors.

■ **Note** Remember to clone the database structure from the development to the test database by executing `rake db:test:clone_structure`. You can also perform the migration by executing the `rake` command without parameters. This will run the tests and the migrations.

Preparing the Models

In Chapter 3, we created the `Book` model. Before creating the `Tag` model, we need to modify the `Book` model so that it can be tagged. This is a simple operation. Just add the `acts_as_taggable` method call to the model, as shown here:

```
class Book < ActiveRecord::Base
  acts_as_taggable
```

This gives us access to the `acts_as_taggable` API methods listed in Tables 7-1 and 7-2. We can now do things like `Book.find_by_title('The Satanic Verses').tag('Novel, Blasphemous')` and `Book.find_tagged_with(:any => 'Blasphemous')` with the model. As you can see, the code reads almost like a sentence written in English.

Next, create the ActiveRecord model for the `tags` table by executing the `generate` script:

```
$ script/generate model Tag --skip-migration
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/tag.rb
      create  test/unit/tag_test.rb
      create  test/fixtures/tags.yml
```

We tell the generate script not to generate a migration file, since we already created it manually. You can open the app/models/tag.rb and examine it. You should see the following code:

```
class Tag < ActiveRecord::Base
end
```

The Tag model doesn't include a mapping to the Book model. This means you can't access the books that are associated with a tag by calling, for example, Tag.find(1).books. This won't be a problem, as we will use the acts_as_taggable API instead, which does the same work with the Book.find_tagged_with method. However, if you need it, you can add the mapping by adding has_and_belongs_to_many :books to the Tag model.

Unit Testing the Model

Upgrading the acts_as_taggable gem or changing your own code can easily break the tagging functionality. To prevent this, or at least minimize the risk of this happening in our production environment, we'll create unit tests for the model. These tests will be used to verify that we can add tags to a book and find the book.

Open test/unit/book_test.rb in your editor and add the following test_tagging method:

```
require File.dirname(__FILE__) + '/../test_helper'

class BookTest < Test::Unit::TestCase
  fixtures :publishers, :authors, :books, :authors_books

  def test_tagging
    book = Book.find(1)
    book.tag 'Elvis,Thriller', :separator => ','

    book.reload

    assert book.tagged_with?('Elvis')
    assert book.tagged_with?('Thriller')
    assert_equal 2, book.tags.size
    assert_equal ['Elvis', 'Thriller'], book.tag_names

    assert_equal 1, Book.find_tagged_with(:any => [ 'Elvis', 'Thriller' ]).size
    assert_equal 1, Book.find_tagged_with(:all => [ 'Elvis', 'Thriller' ]).size
  end
  def test_failing_create
```

The unit test first loads a single book from the `books` fixture and assigns the tags `Elvis` and `Thriller` to it. Then the test reloads the book from the database and verifies that the book has been tagged correctly. The verification is done by using the `acts_as_taggable` API methods `tagged_with` and `find_tagged_with`.

■ **Tip** You can use `book.errors.full_messages.join(';')` to get a string representation of all validation errors after an unsuccessful save of an ActiveRecord object. This is useful when debugging validations.

Next, run the unit test, and you should see all tests pass:

```
$ ruby test/unit/book_test.rb
```

```
Loaded suite test/unit/book_test
Started
.....
Finished in 0.159311 seconds.

5 tests, 22 assertions, 0 failures, 0 errors
```

Using the Console to Test the Model

You normally test an application with your browser or with unit, integration, and functional tests. The `console` script, located in your application's `script` directory, provides you with one more option. When executed, the script loads your application and opens an interactive session that you can use to write and execute code directly in the console window. You can create, update, and delete objects and access your controllers. This is useful when you want to perform additional testing.

■ **Tip** By default, the `console` script loads your development environment. But you can also use it on your production machine, for example, for doing maintenance work on your database. Simply execute `script/console production` and write some code that uses your ActiveRecord models as you would in a controller.

Let's use the console to double-check that tagging works. Open a command prompt and start the Emporium console by executing `script/console` in the application's root directory. You can run `script/console --help` to get a list of options.

When the console has loaded, you should see the following:

```
$ script/console
```

```
Loading development environment.
>>
```

The console is now waiting for input, and you can write the following code in the console to create a publisher, author, and a book:

```
publisher = Publisher.create(:name => 'A Publisher')
author = Author.create(:first_name => 'An', :last_name => 'Author')
book = Book.create(
  :title => 'A Book',
  :publisher => publisher,
  :authors => [author],
  :published_at => Time.now,
  :isbn => '123-123-123-x',
  :blurb => 'The blurb',
  :page_count => 300,
  :price => 30.5
)
```

Press Enter to execute the code and print out the object to the console, as follows (this is a partial listing):

```
=> #<Book:0xb745b42c @new_record=false, @authors=[#<Author:0xb748d314 @new_record...
```

You can now use the `book` object that was created and tag it with the code shown here:

```
book.tag('A B C D E F')
book.save
```

Recall that the `save` method should return `true` if the save was successful. You can verify that the book was tagged correctly by executing a search for books that have been tagged with either A or B:

```
Book.find_tagged_with(:any => 'A B')
```

```
=> [#<Book:0xb73f0820 @attributes={"isbn"=>"123-123-123-x", "updated_at"=>"2006...
```

You should see the method print out to the console the object that you created.

Implementing the User Stories

Web applications like Flickr and del.icio.us allow their users to tag content by entering a list of tags, separated by spaces, in a text field; for example, they may enter `News Europe London`. This is a simple and elegant way of tagging, and is easy enough for George to understand and use. To make George's life even easier, we'll use autocompletion to help him remember the tags that have previously been assigned to books. When George types in, for example, the string `Beer` in the Tags field, the system should display all tags containing that string, including `Beer Tasting` and `Brewing Beer`. This will prevent George from entering duplicate tags.

Implementing the Assign Tags User Story

According to our requirements, George should be able to assign tags to a book on the add book page. We'll modify the page and add an autocompletion field that uses the `script.aculo.us` JavaScript library, which we introduced in Chapter 5.

Updating the Integration Test

First, we'll modify the integration test and add a test that simulates George entering the tags `Elvis`, `Thriller`, and `Cooking` in the Tags field. Change the `test_book_administration` method in `test/integration/book_test.rb` as follows (changes are marked in bold):

```
george = new_session_as(:george)
george.add_book :tags => 'Ruby, Programming, Dummies', :book => {
  :title => 'Ruby for Dummies',
  :publisher_id => publisher.id,
  :author_ids => [author.id],
  :published_at => Time.now,
  :cover_image => fixture_file_upload('/books.yml', 'image/png'),
  :isbn => '123-123-123-X',
  :blurb => 'The best book released since "Eating for Dummies"',
  :page_count => 123,
  :price => 40.4
}
```

The change adds the `tags` parameter, and the specified tags, to the request. This parameter will be used by the controller to tag the book. Note that we can't put it in the `book` hash, as that would make Rails try to assign a string to the `tags` attribute, which it expects to be an array of `Tag` objects.

We want the test to fail in TDD fashion, but if you try to run it now, it will succeed. This is because we haven't yet added a check that verifies that the book was tagged.

Update the add_book method as follows (changes are marked in bold):

```
    assert_response :success
    assert_template "admin/book/list"
    assert_tag :tag => 'td', :content => parameters[:book][:title]
    book = Book.find_by_title(parameters[:book][:title])
    assert_equal parameters[:tags].split(',').size, book.tags.size
    return book
  end
```

With the change, we check that the book can be found by its title and that it is tagged. If you run the test now, it will fail and print out the following message:

```
<3> expected but was
<0>.
```

We haven't created the code that actually tags the book, so the test fails when it checks the size of the tags collection. This is easily fixed by changing the create action in the controller, as follows:

```
  def create
    @book = Book.new(params[:book])
    @book.tag(params[:tags], :separator => ',')

    if @book.save
      flash[:notice] = 'Book was successfully created.'
      redirect_to :action => 'list'
    else
```

The create action calls the book object's tag method to assign the tags. Recall that the tags are specified with the tags parameter, and that the comma is used as a separator. Note that if the user doesn't enter any tags, this will do nothing.

Run the integration test again with the following command:

```
$ ruby test/integration/book_test.rb
```

This time, it should pass without errors.

Modifying the View

Next, we need to change the view. As we explained earlier, the view should have an autocompletion field where George can enter the tags. This functionality could easily be implemented with Ajax, but instead, we'll do it the old-school way with plain JavaScript—just to save some resources and a couple of Ajax calls to the server.

The tags will be stored in a JavaScript variable and passed to the Autocompleter.Local helper provided by script.aculo.us.

■ **Tip** The documentation for `Autocompleter.Local` can be found on the `script.aculo.us` wiki at `http://wiki.script.aculo.us/scriptaculous/show/Autocompleter.Local`.

Next, add the new Tags field to the view file `app/views/admin/book/_form.rhtml`, right after the Title field:

```
<p>
<label for="tags">Tags</label><br/>
<input type="text" id="tags" autocomplete="off" size="50" name="tags" value=" ➥
<%= @book.tags.collect{|tag| tag.name }.join(",") if @book.tags -%>">
</p>
<div id="tags_update"></div>
<script type="text/javascript" language="javascript" charset="utf-8">
// <![CDATA[
  new Autocompleter.Local('tags','tags_update',
  new Array(<%= @tags.collect{|tag| "\"" + tag.name + "\"" }.join(",") %>), ➥
{ tokens: new Array(',','\n'), fullSearch: true, partialSearch: true });
// ]]>
</script>
```

Notice that we specify the tag separator to be a comma with the `tokens` parameter. This is an array and can contain more than one separator. For example, it could also contain the newline character `\n`. Also notice that the tags will be shown in the `tags_update div`.

Modifying the Controller

You can now try to access the page with your browser. It should generate the following error, as the view expects to find the tags instance variable:

```
You have a nil object when you didn't expect it!
```

Fix this by changing the `load` method in the book controller (`app/controllers/admin/book_controller.rb`), as follows:

```
  def load_data
    @authors = Author.find(:all)
    @publishers = Publisher.find(:all)
    @tags = Tag.find(:all)
  end
```

Recall that we use the `load_data` method to load all authors, publishers, and tags in the `new` and `edit` actions. Now, if you open `http://localhost:3000/admin/book/new`, you should see the new field.

Add a book and type in a few tags, such as `Programming, Rails, Ruby`. Save the book, and create another book. Type the character *R* in the Tags field. This time, you should see a list of tags containing both Rails and Ruby.

■ **Note** As stated earlier, this example does not use Ajax. Instead, the tags are all stored in a JavaScript variable. Using Ajax would be easier. The autocompletion text field could be generated with the following code: `<%= text_field_with_auto_complete :book, :tags %>` This would also require us to add the `auto_complete_for_book_tags` action to the controller, which would render a partial view.

Changing the Style Sheet

The list that displays the tags has a white background, which makes it difficult to tell apart from the rest of the page. Remember that the autocompletion field uses a `div` element to show the list of tags. The `div` has been assigned the `id` `#tags_update`, so it is easily fixed by adding the following style to `public/stylesheets/style.css`:

```
#tags_update {
  font-size: 75%;
  font-weight: normal;
  color: white;
  margin: 0px;
  padding: 0px;
  border:1px solid black;
  background-color: #B36B00;
}
```

Also add the following style to highlight a tag when you move your mouse over it:

```
#tags_update li:hover {
  text-decoration: underline;
}
```

Reload the page and add a new book again. This time, you should see the tags in a brown box, as shown in Figure 7-2. You can double-click a tag to add it to the list.

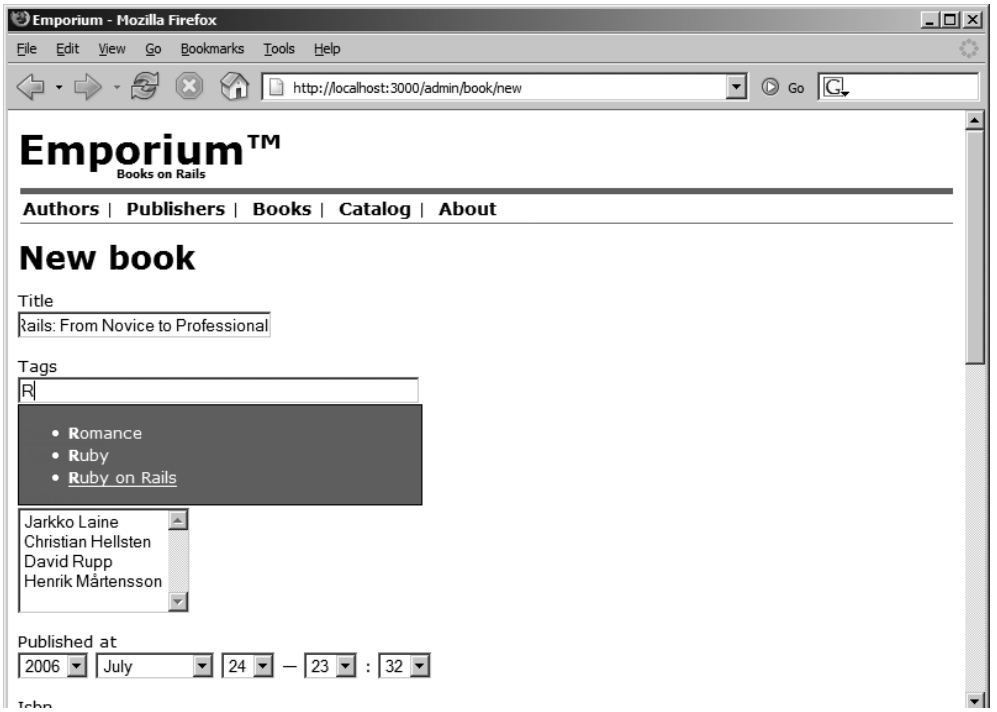


Figure 7-2. *The Tags field with autocompletion*

Implementing the Edit Tags User Story

George must be able to go back to a book that he has added to the system and edit the details, including removing and adding tags. The edit book page uses the same `_form.rhtml` partial as we used for the add book page. This means that most of the work has already been done in the previous section. The only part of the code we need to change is the controller. As usual, we will start by creating the integration test.

Updating the Integration Test

Change the `test_book_administration` method in the integration test `test/integration/book_test.rb`. Add the `tags` parameters to the line that starts with `george.edit_book`, as shown here:

```
george.edit_book(ruby_for_dummies, :tags => 'Toddlers', :book => {
  :title => 'Ruby for Toddlers',
  :publisher_id => publisher.id,
  :author_ids => [author.id],
  :published_at => Time.now,
  :isbn => '123-123-123-X',
  :blurb => 'The best book released since "Eating for Toddlers"',
  :page_count => 123,
  :price => 40.4
})
```

Also change the implementation of `edit_book` DSL method as follows:

```
def edit_book(book, parameters)
  get "/admin/book/edit/#{book.id}"
  assert_response :success
  assert_template "admin/book/edit"

  post "/admin/book/update/#{book.id}", parameters
  assert_response :redirect
  follow_redirect!
  assert_response :success
  assert_template "admin/book/show"

  book.reload
  assert_equal parameters[:tags].split(',').size, book.tags.size
end
```

Note that we reload the `book` object from the database, before we check that the tags have been updated.

Run the test, and you should see it fail with the following error message:

```
<1> expected but was
<3>.
```

You get the error because we haven't yet modified the controller.

Modifying the Controller

Next, change the `edit` action in `app/controllers/admin/book_controller.rb` as follows:

```
  def update
    @book = Book.find(params[:id])
    @book.tag(params[:tags], :separator => ',', :clear => true)

    if @book.update_attributes(params[:book])
      flash[:notice] = 'Book was successfully updated.'
```

By setting the `:clear` parameter to `true`, we first empty the collection of tags, before assigning the new one.

Run the integration tests with `ruby test/integration/book_test.rb`. This time, all tests should pass without errors.

Modifying the Views

We have finished the implementation of this user story. Now, we need to check that it works by editing an existing book that has been assigned some tags. Add or remove a tag and verify the results. Notice that there's one small problem we should fix before moving on to the next user story: the book details and book list pages should display the tags to the administrator.

Start by changing the book list view. Open `app/views/admin/book/list.rhtml` in your editor and modify the existing code as shown here:

```
  <tr>
    <th><a href="?sort_by=publisher_id">Publisher</a></th>
    <th><a href="?sort_by=title">Title</a></th>
    <th>Tags</th>
    <th><a href="?sort_by=isbn">ISBN</a></th>
    <th colspan="3"></th>
  </tr>

<% for book in @books %>
  <tr>
    <td><%= book.publisher.name %></td>
    <td><%= link_to book.title, :action => 'show', :id => book %></td>
    <td><%= display_tags book %></td>
    <td><%= book.isbn %></td>
    <td><%= link_to 'Show', :action => 'show', :id => book %></td>
```

Notice that we have introduced a new helper method called `display_tags`, because we need to display the tags on both pages. Instead of duplicating the code in the view, we add the following method to the application helper (`app/helpers/application_helper.rb`):

```
def display_tags(book)
  book.tags.collect{|tag| tag.name }.join(", ") if book.tags
end
```

By adding the new method to the application helper, we can now access it from any view in our application.

After making the change, access `http://localhost:3000/admin/book/list`. You should now see the tags displayed in the list, as shown in Figure 7-3.

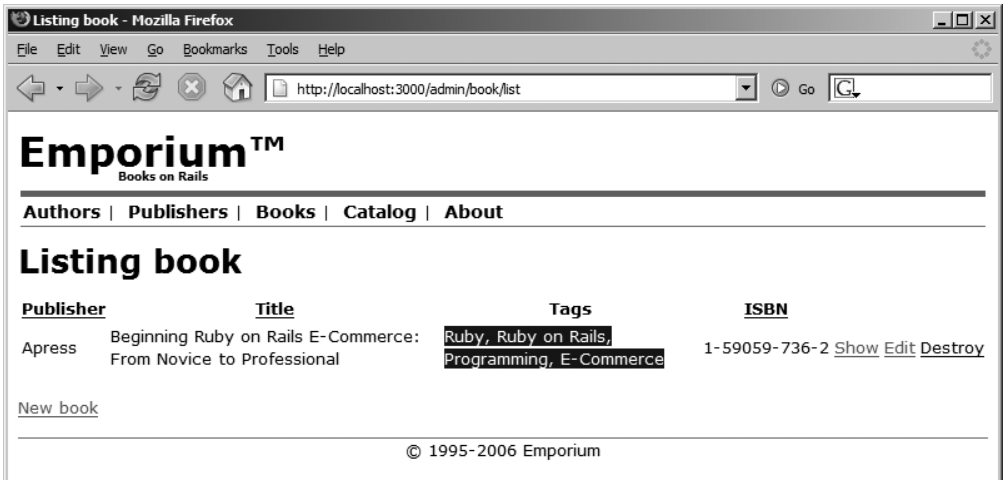


Figure 7-3. *The book list page showing the Tags column*

Next, change the book details view file. Open `app/views/admin/book/show.rhtml` and add the following after the Title field:

```
<dt>Tags</dt>
<dd><%= display_tags @book %></dd>
```

Now, when you access the book details page, you should see the tags on the screen, as shown in Figure 7-4.

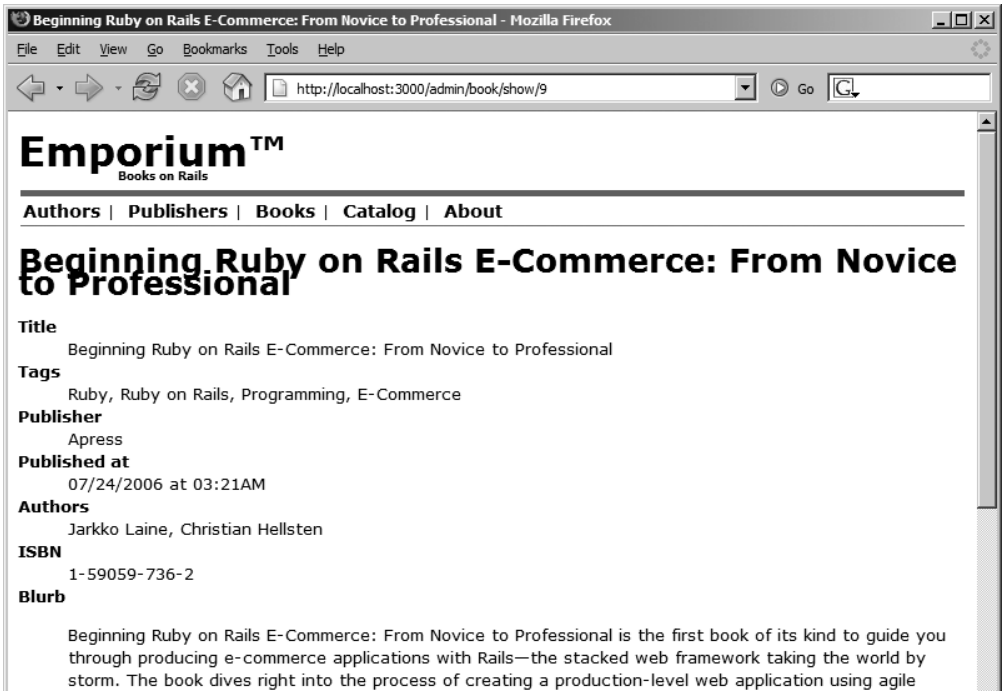


Figure 7-4. *The book details page showing the book's tags*

We are now confident that the administrator's part of the tagging functionality works as George wants. We tell him to come over and have a look. Again, he seems happy that it is so simple and that it works like a train on rails.

Implementing the List Tags and Show Tag User Stories

The Show Tag user story describes how customers should be able to view all tags in the system. Before we continue with the next user story implementation, George reminds us that he expects us to complete the three remaining user stories today: "I have friends in Bangalore! I'll just offshore the whole project to India, if you can't get it done today."

To speed up development, we'll use the `generate` script to create a new controller. Execute the `generate` script with the following parameters to generate the controller for these user stories:

```
$ script/generate controller tag list show
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/tag
      exists  test/functional/
      create  app/controllers/tag_controller.rb
      create  test/functional/tag_controller_test.rb
      create  app/helpers/tag_helper.rb
      create  app/views/tag/list.rhtml
      create  app/views/tag/show.rhtml
```

This creates the controller, functional test, helper, and views. The List tags and Show tag user stories are easy to implement, as we'll show you next

First, open the `tag_controller.rb` file, and change the `list` and `show` actions as shown here:

```
  def list
   @page_title = 'Listing tags'
   @tag_pages, @tags = paginate :tags, :order => :name, :per_page => 10
  end

  def show
    tag = params[:id]
    @page_title = "Books tagged with '#{tag}'"
    @books = Book.find_tagged_with(:any => tag, :separator => ',')
  end
```

The `list` action uses the standard `paginate` helper to show a paginated list of tags. The `show` action uses the `find_tagged_with` method to find books having the specified tag.

Next, add the following code to `app/views/admin/books/list.rhtml`:

```
<% for tag in @tags %>
  <%= link_to tag.name, :action => 'show', :id => tag.name %><br/>
<% end %>
<p>
<%= link_to 'Previous page', { :page => @tag_pages.current.previous } ➥
if @tag_pages.current.previous %>
<%= link_to 'Next page', { :page => @tag_pages.current.next } ➥
if @tag_pages.current.next %>
</p>
```

The view loops through the tags and generates a link to the show tag page. At the bottom of the page we show the pagination links.

For the show tag page, we need to add the following to `show.rhtml`:

```
<% for book in @books %>
  <%= link_to book.title, :controller => 'catalog', :action => 'show', ➥
:id => book %><br/>
<% end %>
```

This simply loops through all the books and links to the book details page we implemented in Chapter 4. Now you can do a quick test by accessing `http://localhost:3000/tag/list`. You should see a list of tags, as shown in Figure 7-5.

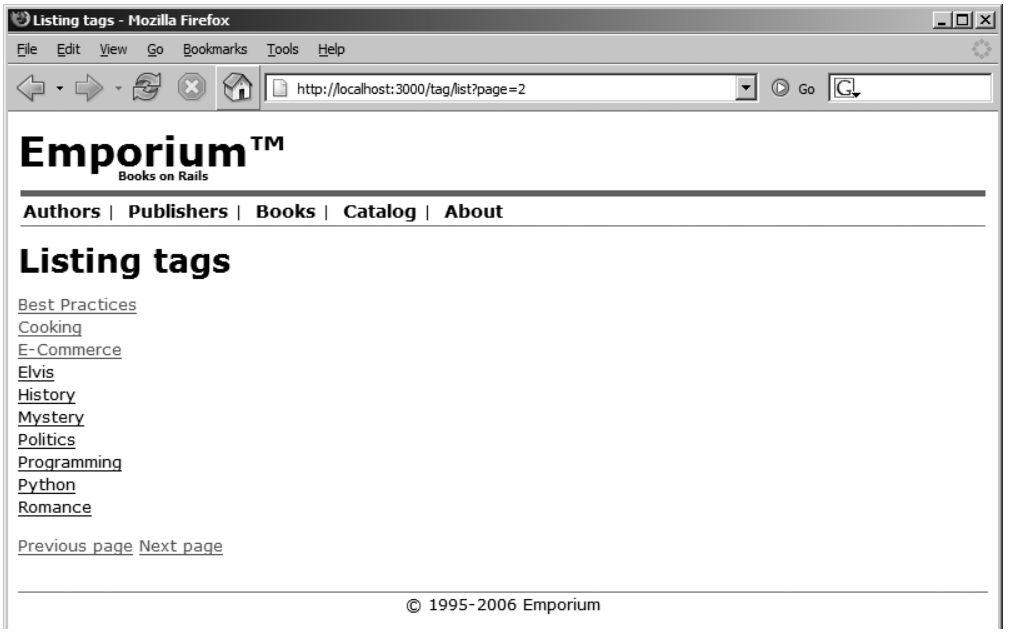


Figure 7-5. *The tag list page showing a list of tags*

If you click a tag, you should see all books that have been tagged with that specific tag, as shown in Figure 7-6. Clicking the book title takes you to the book details page.

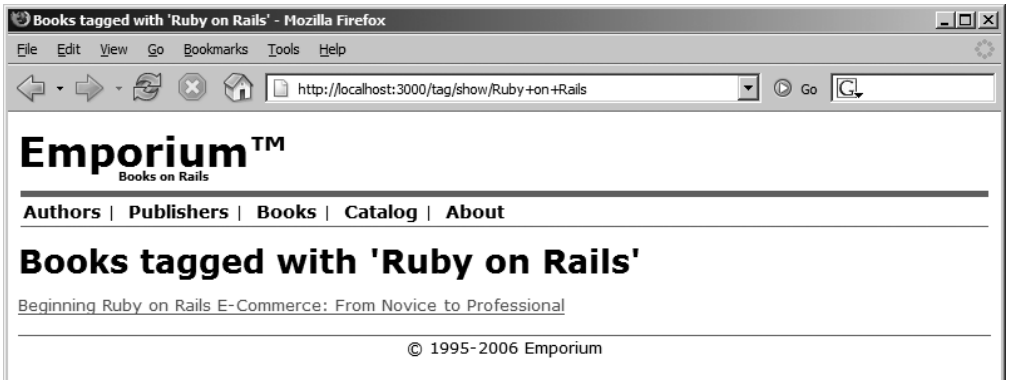


Figure 7-6. *The show tag page*

We still don't have a link to either the tag list or the show tag page. It would be natural to link to the show tag page from all the places where the tag is being displayed. Luckily, we used a helper to display the list of tags that have been assigned to a book, so the change is just in one place. Change the `display_tags` method in `app/helpers/application_helper.rb` as follows:

```
  def display_tags(book)
    book.tags.collect{|tag| link_to tag.name, :controller => '/tag', ➥
:action => 'show', :id => tag.name }.join(", ") if book.tags
  end
```

Instead of just showing the tag's name, we now link to the show tag page.

■ **Note** Notice that we needed to prepend a forward slash to the controller, so that the admin pages also point to the correct page. If we had used only `:controller => 'tag'`, the generated URL would be `/admin/tag/show/x`.

Now, if you access any of the pages where we display a book and the tags, the tags should be clickable and point to the show tag page.

A good place for putting a link to the tag list page would be the menu. To do this, add the following code to `application.rhtml`:

```
<li><a href="/admin/book">Books</a> | </li>
<li><a href="/tag/list">Tags</a> | </li>
<li><a href="/">Catalog</a> | </li>
<li><a href="/about">About</a> </li>
```

Implementing the Recommend Books User Story

Recall that the Recommend Books user story describes how George wants to be able to automatically recommend related books to customers. The `acts_as_taggable` gem has two methods that we can use for implementing this functionality:

- `book.tagged_related`: We can use this instance method to display books that share one or more of the same tags, and that are related to the current book being displayed to the customer.
- `Book.find_related_tags`: We can use this class method to recommend books that use tags that are related to the tags used by the currently displayed book.

Open app/views/catalog/show.rhtml in your editor and add the following code to it:

```
<dl>
  <dt>Price</dt>
  <dd>$<%= sprintf("%0.2f", @book.price) -%></dd>
  <dt>Page count</dt>
  <dd><%= @book.page_count -%></dd>
  <dt>Tags</dt>
  <dd><%= display_tags @book -%></dd>
</dl>

<% if @book.tags.size > 0 %>
<div id="recommended">
<h2>Recommendations</h2>
<h4>Books</h4>
<% for book in @book.tagged_related %>
  <%= link_to book.title, :action => 'show', :id => book.id %><br/>
<% end %>

<h4>Tags</h4>
<% for tag in Book.find_related_tags(@book.tags.collect(&:name), ➥
:separator => ',', :raw => true, :limit => 100) %>
  <%= link_to tag['name'], :controller => 'tag', :action => 'show', ➥
:id => tag['name'] %><br/>
<% end %>
</div>
<% end %>
```

We now show the tags on the page. We also added a div that is used to recommend similar books and tags to the customer. This div uses the id recommended, so that we can style it with CSS. We want it to be displayed as a sidebar to the right of the page. We do this by adding the following to public/stylesheets/style.css:

```
#recommended {
  border-left: 3px solid #666;
  background-color: white;
  position: fixed;
  bottom: 0;
  right: 0;
  width: 200px;
  height: 100%;
  padding: 5px 10px;
}
```

Note that the recommendations are done with the `acts_as_taggable` API.

Now add three new books with tags, as follows:

- For the first book, add the tags `Ruby, Ruby on Rails, Programming`.
- For the second book, add the tags `Ruby, Programming`.
- For the third book, add the tags `Ruby, Ruby on Rails, Programming, E-Commerce`.

You should see something similar to Figure 7-7 when accessing the second book. The system automatically recommends the first and third book, and the `Ruby on Rails` and `E-Commerce` tags to the customer.

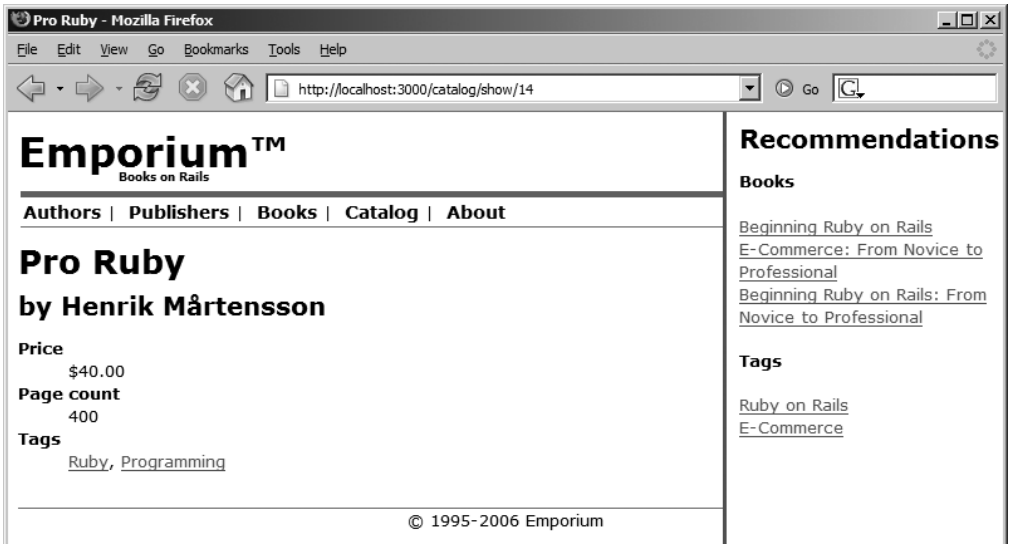


Figure 7-7. *The book details page displaying recommendations in the sidebar*

Related books are pulled out of the database by calling `book.tagged_related`, which returns an array of books. Related tags are displayed with the code shown here:

```
<% for tag in Book.find_related_tags(@book.tags.collect(&:name), ➥
:separator => ',', :raw => true) %>
  <%= link_to tag['name'], :controller => 'tag', :action => 'show', ➥
:id => tag['name'] %><br/>
<% end %>
```

We specify three parameters for the `Book.find_related_tags` method:

- `@book.tags.collect(&:name)`: This is shorthand notation for `@book.tags.collect(|tag| tag.name)`, and returns an array of tag names.
- `:separator => ','`: We specify that the separator is a comma.
- `:raw => true`: We specify that we want an array of hashes returned, so that we can use `tag['name']` instead of `tag[0]`, to access the tag names.

With five minutes left in the workday, we call in George to do some acceptance testing of the work done in this sprint. He's happy with the results and tells us that he's impressed at how incredibly fast we were able to finish the tagging functionality. He says, "That other consultant told me it would take a month to complete, and you do it in one day!

Summary

In this chapter, we showed you how to implement a tagging system that allows you to add tags to books and later edit them. Using the `acts_as_taggable` gem, we built a system that is able to recommend similar or related items to online shoppers. Along the way, we showed you how to use the console to test the model, write appropriate integration tests for the new functionality, and implement an autocompletion field using the `script.aculo.us` JavaScript library.

In the next chapter, we'll secure our application.

CHAPTER 8

Security

Our application is already fairly extensive. George can administer all kinds of things in the application, including books, authors, and publishers. However, the application has one major shortcoming: Anyone can browse to the administrative part of the site and wreak havoc by deleting and editing information.

In this chapter, we will show you how to implement a basic authentication system for an application with the help of the `acts_as_authenticated` plugin. We will also take a look at some common security problems in web applications and give you tips on how to use Rails to avoid them.

Getting the Authentication Requirements

We need to support three basic scenarios in the Emporium's authentication system:

- *Log in*: George has just gotten his hands on Henrik Mårtensson's *Pro Ruby*, and absolutely wants to add it to his catalog. However, as he hasn't logged in already, when he tries to access the admin section of the site, he is redirected to a login page. George gives his credentials and is automatically redirected to the add book page, where he tried to go in the first place.

- *Fail log in*: While George is busy maintaining his catalog, another guy tries to access the admin pages, too. His name is Dirty Harry and his intentions are too evil to print here. Luckily for George, Harry doesn't know the admin username and password. Harry is redirected to the login page, just as George is. Here, he tries to log in with scott/tiger, so his attempts fail, and he is just shown the login form with an error message each time.

- *Reset password*: George has an amazing memory. It's just sometimes a bit short. Thus, occasionally, he forgets his password to the system. Then he just clicks a link to reset his password, and the system sends him the new one by e-mail. After he has received the new password, he can again log in to the system successfully.

Once we put together the authentication system, George will sleep a lot more peacefully—he won't need to worry about people wreaking havoc on the Emporium site.

Using the Authentication Plugin

We can create a simple authentication framework for our Rails application by using the `acts_as_authenticated` plugin (`http://technoweenie.stikipad.com/plugins/show/Acts+as+Authenticated`), written by Rails core team member Rick Olson.

Let's start by installing the plugin in our application. Enter the following command to tell the Rails plugin framework to look for plugins in the given repository:

```
$ script/plugin source http://svn.techno-weenie.net/projects/plugins
```

```
Added 1 repositories.
```

Next, run the actual `install` command:

```
$ script/plugin install acts_as_authenticated
```

```
+ ./acts_as_authenticated/CHANGELOG
+ ./acts_as_authenticated/README
+ ./acts_as_authenticated/generators/authenticated/USAGE
... many lines omitted ...
Consult the Acts As Authenticated wiki for more:
http://technoweenie.stikipad.com/plugins/show/Acts+as+Authenticated
```

Now that the plugin is installed, the next step is to generate the models and controllers for authentication. The plugin installs custom generators just for this, so all we need to do is to execute the following command:

```
$ script/generate authenticated user account
```

```
exists  app/models/
exists  app/controllers/
exists  app/helpers/
create  app/views/account
exists  test/functional/
exists  test/unit/
create  app/models/user.rb
create  app/controllers/account_controller.rb
create  lib/authenticated_system.rb
create  lib/authenticated_test_helper.rb
create  test/functional/account_controller_test.rb
create  app/helpers/account_helper.rb
create  test/unit/user_test.rb
create  test/fixtures/users.yml
create  app/views/account/index.rhtml
create  app/views/account/login.rhtml
create  app/views/account/signup.rhtml
exists  db/migrate
create  db/migrate/009_create_users.rb
```

As you can see from the output, the `generate` command created the following:

- A new model named `User` and a new controller named `AccountController`, as well as tests for both of them
- The views for the login functionality and a new module containing the authentication code, `AuthenticatedSystem`, in the `lib` directory
- A new migration (`db/migrate/009_create_users.rb`) to bring the new user model into the database, shown in Listing 8-1

Listing 8-1. *ActiveRecord Migration for the Users Table*

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table "users", :force => true do |t|
      t.column :login,                     :string
      t.column :email,                     :string
      t.column :crypted_password,          :string, :limit => 40
      t.column :salt,                      :string, :limit => 40
      t.column :created_at,                :datetime
      t.column :updated_at,                :datetime
      t.column :remember_token,            :string
      t.column :remember_token_expires_at, :datetime
    end
  end

  def self.down
    drop_table "users"
  end
end
```

Notice that the password will be stored in the database in an encrypted form.

Now let's run the migration to get our database up-to-date. (Don't forget to run `rake db:test:clone_structure` afterwards to clone the new additions to the test database, too.)

```
$ rake migrate
```

```
(in /home/george/projects/emporium)
== CreateUsers: migrating =====================================================
-- create_table("users", {:force=>true})
   -> 0.2946s
== CreateUsers: migrated (0.2953s) ============================================
```

Great! We now have a working authentication framework deployed in our system.

If you take a look at the beginning of the new `AccountController` in `app/controllers/account_controller.rb`, you can see that `AuthenticatedSystem` is mixed in the controller:

```
class AccountController < ApplicationController
  # Be sure to include AuthenticationSystem in Application Controller instead
  include AuthenticatedSystem
  ...
```

However, we want the authentication system to be available to other controllers as well, so let's move the `include` line from `AccountController` to `ApplicationController` in `app/controllers/application.rb`:

```
class ApplicationController < ActionController::Base
  include AuthenticatedSystem

  private

  def initialize_cart
  ...
```

As `ApplicationController` is the parent class of all our controllers, authentication functionality is now provided throughout our application. It's only a matter of putting it into action where necessary.

Since we want to make the tests provided by the plugin work as well, we also move the following line from the `AccountControllerTest` class in `test/functional/account_controller_test.rb` to the beginning of the `Test::Unit::TestCase` class in `test/test_helper.rb`:

```
  include AuthenticatedTestHelper
```

With our authentication framework in place, we're ready to implement our authentication user stories.

Implementing the User Stories

As usual, we will take the TDD approach while implementing the user authentication system. For this sprint, we will use integration tests, as we have done in previous chapters.

Implementing the Log In User Story

We start the grunt work by creating a new integration test case for the login functionality.

```
$ script/generate integration_test authentication
```

```
      exists  test/integration/
      create  test/integration/authentication_test.rb
```

First, we want to test that when George tries to go to the admin section of the site, he gets redirected to the login page. Open `test/integration/authentication_test.rb` and create the DSL for our integration test, as shown in Listing 8-2.

Listing 8-2. *First Version of the Authentication Integration Test*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class AuthenticationTest < ActionController::IntegrationTest
  def test_successful_login
    george = enter_site(:george)
    george.tries_to_go_to_admin
  end

  private

  module BrowsingTestDSL
    include ERB::Util
    attr_writer :name

    def tries_to_go_to_admin
      get "/admin/book/new"
      assert_response :redirect
      assert_redirected_to "/account/login"
    end

  end

  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

Here, the most interesting part is in the `tries_to_go_to_admin` method. This is where we test that the first part of the story goes as planned: George is redirected to the login page when trying to access admin pages. If you run the test, you get the following failure:

```
$ ruby test/integration/authentication_test.rb
```

```
Loaded suite test/integration/authentication_test
Started
F
Finished in 1.44942 seconds.

  1) Failure:
test_successful_login(AuthenticationTest)
    [test/integration/authentication_test.rb:17:in 'tries_to_go_to_admin'
     test/integration/authentication_test.rb:6:in 'test_successful_login'
     /usr/local/lib/ruby/gems/1.8/gems/actionpack-
1.12.1/lib/action_controller/integration.rb:427:in 'run']:
Expected response to be a <:redirect>, but was <200>

1 tests, 1 assertions, 1 failures, 0 errors
```

It seems the redirection is not working, which should come as no surprise. Now it's time to put the authentication plugin to work.

Adding the Filter

In Chapter 5, we hinted that filters would be a good fit for implementing authentication functionality in Rails, and that is exactly what `acts_as_authenticated` does (or, to be precise, makes us do). The `AuthenticatedSystem` module (which is now included in all our controllers, remember?) implements a function called `login_required`, which is the workhorse of the whole plugin. If it's called as a `before_filter` inside a controller, a login check is done before any action in that controller is let loose:

```
class SomeController < ApplicationController
  before_filter :login_required

  def first_action
    # this action is now only available for logged in users
  end
end
```

As you might remember from Chapter 5, you can restrict the filter to certain actions by using the `:only` and `:except` parameters in the `before_filter` call:

```
before_filter :login_required, :only => :secret_action
before_filter :login_required, :except => [:index, :rss]
```

In our case, we want to protect all controllers in the app/controllers/admin directory. This is most easily done by creating a common parent class for them:

```
$ script/generate controller 'admin/base'
```

Next, we'll put the filter macro in the newly created app/controllers/admin/base_controller.rb file:

```
class Admin::BaseController < ApplicationController
  before_filter :login_required
end
```

Now we need to make the actual admin controllers inherit from Admin::BaseController. Make the following change in all controller files (for authors, books, and publishers) in app/controllers/admin, except the one we just created:

```
class Admin::AuthorController < Admin::BaseController
```

Note that the classes are still descendants of ApplicationController because Admin::BaseController inherits it.

If you now run the integration test file again, you'll see that the protection works as it should:

```
$ ruby test/integration/authentication_test.rb
```

```
Loaded suite test/integration/authentication_test
Started
.
Finished in 0.20896 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
```

Testing Redirection

The last part of the story was that after successful login, George is redirected to the page he tried to access in the first place. acts_as_authenticated should do this for us automatically. Let's extend our integration test (authentication_test.rb) as follows to make sure.

```
require "#{File.dirname(__FILE__)}/../test_helper"

class AuthenticationTest < ActionController::IntegrationTest
  def setup
    User.create(:login => "george",
                :email => "george@emporium.com",
                :password => "cheetah",
                :password_confirmation => "cheetah")
  end
```

```
  def test_successful_login
    george = enter_site(:george)
    george.tries_to_go_to_admin
    george.logs_in_successfully("george", "cheetah")
  end

  private

  module BrowsingTestDSL
    include ERB::Util
    attr_writer :name

    def tries_to_go_to_admin
      get "/admin/book/new"
      assert_response :redirect
      assert_redirected_to "/account/login"
    end

    def logs_in_successfully(login, password)
      post_login(login, password)
      assert_response :redirect
      assert_redirected_to "/admin/book/new"
    end

    private

    def post_login(login, password)
      post "/account/login", :login => login, :password => password
    end
  end


  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

In the beginning of the test, we use the `setup` method, which is automatically run before every test method, to create George as a user in the system. Then we create another DSL method for logging in to the system successfully. We extracted the actual posting of the login credentials to a private method, because we will need the same code later when we test a failed login. All our new method tests is that after successful login, George is redirected to `/admin/book/new`, the page he tried to access before he was thrown to the login page.

Running the test again shows that the authentication system indeed remembers where George was heading:

```
$ ruby test/integration/authentication_test.rb
```

```
Loaded suite test/integration/authentication_test
Started
.
Finished in 0.192056 seconds.

1 tests, 6 assertions, 0 failures, 0 errors
```

Trying to access the admin pages in a browser confirms what the test already says. As you can see in Figure 8-1, if you haven't logged in successfully, you're redirected to the login page.

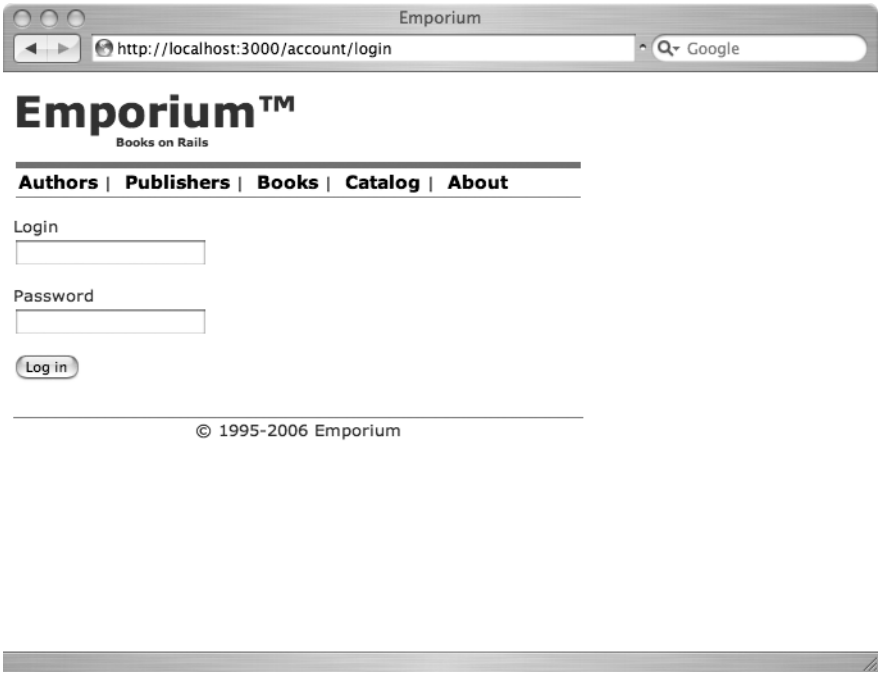


Figure 8-1. *Accessing the admin pages redirects to the login page*

Implementing the Fail Log In User Story

To make sure that logging in with incorrect credentials doesn't work, we can use the same basic code we already have in place in `test/integration/authentication_test.rb`, with only some slight additions to the DSL:

```
require "#{File.dirname(__FILE__)}/../test_helper"

class AuthenticationTest < ActionController::IntegrationTest
  def setup
    User.create(:login => "george",
                :email => "george@emporium.com",
                :password => "cheetah",
                :password_confirmation => "cheetah")
  end

  def test_successful_login
    george = enter_site(:george)
    george.tries_to_go_to_admin
    george.logs_in_successfully("george", "cheetah")
  end

  def test_failing_login
    harry = enter_site(:harry)
    harry.tries_to_go_to_admin
    harry.attempts_login_and_fails("scott", "tiger")
  end

  private

  module BrowsingTestDSL
    include ERB::Util
    attr_writer :name

    def tries_to_go_to_admin
      get "/admin/book/new"
      assert_response :redirect
      assert_redirected_to "/account/login"
    end
```

```
    def logs_in_successfully(login, password)
      post_login(login, password)
      assert_response :redirect
      assert_redirected_to "/admin/book/new"
    end

    def attempts_login_and_fails(login, password)
      post_login(login, password)
      assert_response :success
      assert_template "account/login"
      assert_equal "Incorrect login!", flash[:notice]
    end

    private

    def post_login(login, password)
      post "/account/login", :login => login, :password => password
    end
  end


  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

As you can see, Harry's case is similar to George's, but he tries to log in with an account that doesn't exist. In `attempts_login_and_fails`, we check that he is not redirected and is served the login form again. You can run the test and see that it almost works:

```
$ ruby test/integration/authentication_test.rb
```

```
Loaded suite test/integration/authentication_test
Started
F.
Finished in 0.36565 seconds.

  1) Failure:
test_failing_login(AuthenticationTest)
    [test/integration/authentication_test.rb:45:in 'attempts_login_and_fails'
     test/integration/authentication_test.rb:20:in 'test_failing_login'
     /usr/local/lib/ruby/gems/1.8/gems/actionpack-
1.12.1/lib/action_controller/integration.rb:427:in 'run']:
<"Incorrect login!"> expected but was
<nil>.

2 tests, 12 assertions, 1 failures, 0 errors
```

The failure means that the flash message is not set as we would like it to be. This is something that the plugin lets the developer handle.

Adding the Flash Message

Open `app/controllers/account_controller.rb` and add the flash message to show for failed logins:

```
class AccountController < ApplicationController
  # If you want "remember me" functionality, add this before_filter to➥
Application Controller
  before_filter :login_from_cookie

  # say something nice, you goof!  something sweet.
  def index
    redirect_to(:action => 'signup') unless logged_in? || User.count > 0
  end
```

```
  def login
    return unless request.post?
    self.current_user = User.authenticate(params[:login], params[:password])
    if current_user
      if params[:remember_me] == "1"
        self.current_user.remember_me
        cookies[:auth_token] = { :value => self.current_user.remember_token ,➥
:expires => self.current_user.remember_token_expires_at }
      end
      redirect_back_or_default(:controller => '/account', :action => 'index')
      flash[:notice] = "Logged in successfully"
    end
    flash.now[:notice] = "Incorrect login!"
  end
...
```

Running the test again shows all green, so try it out in the browser. Figure 8-2 shows the result. As you can see, the plugin is doing its job.

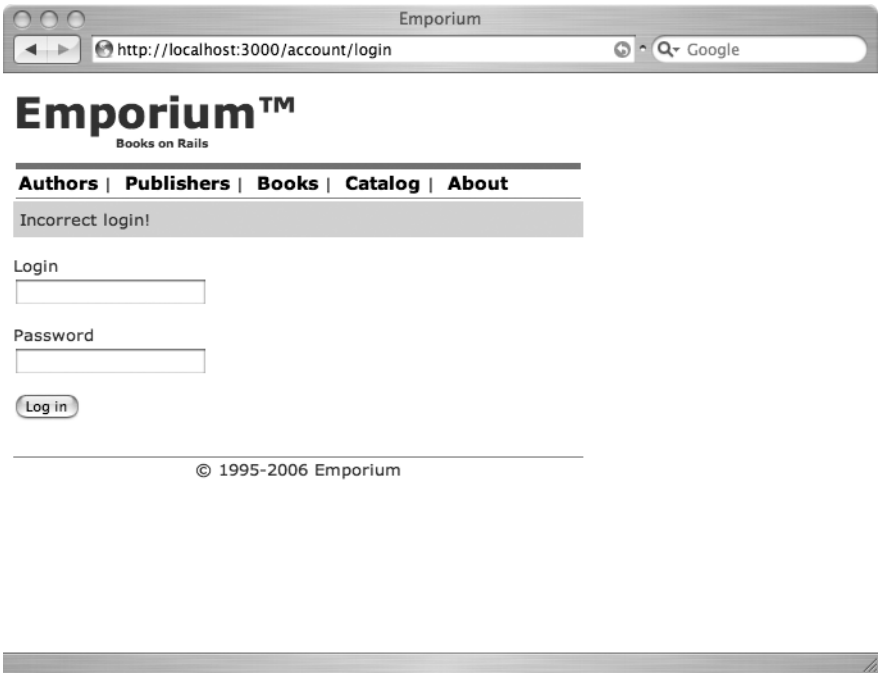


Figure 8-2. *A failed login*

Adding Login Links and Styling

Let's finish off by adding links for logging in and out in the default layout file, `app/views/layouts/application.rhtml`:

```
<div id="header">
  <h1 id="logo">Emporium&trade;</h1>
  <h2 id="slogan">Books on Rails</h2>
  <p id="loginlogout">
    <% if current_user %>
      Logged in as <%= current_user.login %>
      (<%= link_to "Logout", :controller => "/account", :action => "logout" %>)
    <% else %>
      <%= link_to "Login", :controller => "/account", :action => "login" %>
    <% end %>
  </p>
</div>
```

Finally, let's add a bit of styling for the `#loginlogout` box. Open `public/stylesheets/style.css` and add the following rules at the bottom of the file:

```
#loginlogout {
  background-color: #ccc;
  padding: 8px;
  width: 100px;
  position: absolute;
  top: 0px;
  right: 10px;
}
```

Now the login status box appears in the top-right corner of the page, as shown in Figure 8-3.

Figure 8-3. *Styled login status box*

Implementing the Reset Password User Story

To implement the third user story, Reset Password, we need a way to send e-mail messages from our application. The `acts_as_authenticated` plugin comes with a generator for this, too. Execute the following command:

```
$ script/generate authenticated_mailer user
```

```
      exists  app/models/
      create  app/views/user_notifier
      exists  test/unit/
      create  app/models/user_notifier.rb
      create  app/models/user_observer.rb
      create  test/unit/user_notifier_test.rb
      create  app/views/user_notifier/activation.rhtml
      create  app/views/user_notifier/signup_notification.rhtml
```

There are two interesting things created by the `generate authenticated_mailer` command: `UserNotifier` (an ActionMailer object) and `UserObserver` (an observer). Even though neither of them is a normal ActiveRecord model, they both reside in the `app/models` directory. We'll cover the mailer part now and talk about observers after we update the `User` model.

Using ActionMailer Mailers

Rails has a specific package for sending (and receiving, which we don't need in this chapter) e-mail called *ActionMailer*. ActionMailer mailers are Rails classes stored in `app/models` just like normal ActiveRecord models, but they work quite differently.

The `UserNotifier` mailer class we just created in `app/models/user_notifier.rb` looks like this:

```
class UserNotifier < ActionMailer::Base
  def signup_notification(user)
    setup_email(user)
    @subject    += 'Please activate your new account'
    @body[:url]  = "http://YOURSITE/account/activate/#{user.activation_code}"
  end

  def activation(user)
    setup_email(user)
    @subject    += 'Your account has been activated!'
    @body[:url]  = "http://YOURSITE/"
  end

  protected
  def setup_email(user)
    @recipients  = "#{user.email}"
    @from        = "ADMINEMAIL"
    @subject     = "[YOURSITE] "
    @sent_on     = Time.now
    @body[:user] = user
  end
end
```

Here, `signup_notification` and `activation` represent two different e-mail messages sent by the class. The former is sent when a new user has registered and must activate her account, and the latter is sent when the activation is complete. They both use the protected `setup_email` method to prepare common header attributes of the e-mail, such as `recipients`, `from`, and `subject`. You can also set attributes for the message body, such as `@body[:url]` and `@body[:user]`. They will be available as instance variables in the e-mail templates.

We don't need the two mail methods that exist in the mailer, so we delete them and add our own method, as follows:

```
class UserNotifier < ActionMailer::Base
  @@session = ActionController::Integration::Session.new

  def forgot_password(user)
    setup_email(user)
    @subject += "Password reset"
    @body[:url] = @@session.url_for(:controller => "account",
                          :action => "reset_password",
                          :id => user.pw_reset_code, :only_path => false)
  end

  protected
  def setup_email(user)
    @recipients  = "#{user.email}"
    @from        = "admin@emporium-books.com"
    @subject     = "[Emporium] "
    @sent_on     = Time.now
    @body[:user] = user
  end
end
```

`forgot_password` is the mail method we deliver when George or someone from his staff requests a password reset. In the method, we set the subject for the mail, as well as define the password-reset URL sent in the e-mail message. Note that as `url_for` is an instance method for `ActionController` controllers, we can't call it directly from inside a mailer. However, with the trickery on the first line, we create a new `ActionController::Integration::Session` object through which we can call `url_for`, and store it in a class variable, which can be used everywhere inside our mailer class. We also change the `setup_email` method a bit, to accommodate our application.

Next, we need to create a template for the mail body. Create a new template called `forgot_password.rhtml` in `app/views/user_notifier` and put the following code in it:

```
Dear <%= @user.login %>,

Click the following link to reset your password at Emporium:
<%= @url %>
```

As you can see, the `@body` hash contents from the mailer method have been extracted to instance variables in the template, so that, for example, `@body[:user]` became `@user` and `@body[:url]` became `@url`.

Now that we have a mailer class and template ready, we can deliver a password-reset e-mail message by calling `UserNotifier.deliver_forgot_password(@user_object)`. Rails will automatically retrieve the mailer method name after the `deliver_` part in the method call, and deliver the mail prepared by that method.

■ **Tip** If you want to delay the delivery of the e-mail (for example, because you have a mail sweeper that takes care of the deliveries), you can use `create_` instead of `deliver_` in the method call, and you will get a `TMail` object in return. For more information about `TMail`, see `http://i.loveruby.net/en/projects/tmail`.

Updating the User Model

To accommodate resetting a password, we need to add a new field to the `User` model. This field will hold the generated random token that the system will e-mail to George when he forgets his password. Only with this token can he get to a page where he can change to a new password. Run the following code to generate the migration file:

```
$ script/generate migration add_pw_reset_code_to_users
```

```
      exists  db/migrate
      create  db/migrate/010_add_pw_reset_code_to_users.rb
Loaded suite script/generate
```

Now open the new file (`db/migrate/010_add_pw_reset_code_to_users.rb`) and change it to add the new column, as follows:

```
class AddPwResetCodeToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :pw_reset_code, :string, :limit => 40
  end

  def self.down
    remove_column :users, :pw_reset_code
  end
end
```

Run `rake db:migrate` for the changes to take effect.

Next, we need to change the `User` model in `app/models/user.rb` so that we can create a new reset code when needed:

```
require 'digest/sha1'
class User < ActiveRecord::Base
  # Virtual attribute for the unencrypted password
  attr_accessor :password, :password_forgotten

  # ... scroll 'til the end of the file

  def forgot_password
    self.password_forgotten = true
    create_pw_reset_code
  end

  def reset_password
    update_attributes(:password_reset_code => nil)
  end

  protected
    def create_pw_reset_code
      self.pw_reset_code = Digest::SHA1.hexdigest("secret-#{Time.now}")
    end

    # before filter
    def encrypt_password
      return if password.blank?
      self.salt = Digest::SHA1.hexdigest("--#{Time.now.to_s}--#{login}--") ➥
if new_record?
      self.crypted_password = encrypt(password)
    end

    def password_required?
      crypted_password.blank? || !password.blank?
    end
end
```

In the beginning of the file, we declare an instance variable called `@password_forgotten` and accessor methods for it. Then we create a new method, `forgot_password`, which uses this variable to state whether a password reset has been requested. This method sets the `@password_forgotten` variable to `true` using its accessor method, and then calls the protected method `create_pw_reset_code` to create a random, unique 40-character token for this resetting case. `reset_password` will be called when George has successfully completed the process. All it does is set the `password_reset_code` attribute to `nil`, awaiting the next time George's memory shows signs of deterioration.

Using Observers

When we created `UserNotifier` in the beginning of the Reset Password user story implementation, the generator also created a file called `user_observer.rb` in `app/models`. *Observers* in Rails are classes that monitor the life cycle of ActiveRecord objects, somewhat similar to the filters for controllers. Observers support the following callback methods:

- `after_create`
- `after_destroy`
- `after_save`
- `after_update`
- `after_validation`
- `after_validation_on_create`
- `after_validation_on_update`
- `before_create`
- `before_destroy`
- `before_save`
- `before_update`
- `before_validation`
- `before_validation_on_create`
- `before_validation_on_update`

You can call these callbacks directly in an ActiveRecord model, too:

```
class MyModel < ActiveRecord::Base
  after_save :say_foo

  def say_foo
    logger.info "Foo-oo!"
  end
end
```

However, if your callback code gets longer and/or you want to implement similar behavior for multiple models, it's a good idea to extract the callbacks to an observer. Observers also give you more flexibility, since you can restrict the callbacks to happen only in certain controllers, as we will do in this section.

When the `generate authenticated_mailer user` command created the `UserObserver` observer, it created two callbacks for it:

```
class UserObserver < ActiveRecord::Observer
  def after_create(user)
    UserNotifier.deliver_signup_notification(user)
  end

  def after_save(user)
    UserNotifier.deliver_activation(user) if user.recently_activated?
  end
end
```

However, we don't need either of these callbacks, since we aren't implementing signup notification or user activation in this sprint. We can simplify the observer to look like this:

```
class UserObserver < ActiveRecord::Observer
  def after_save(user)
    UserNotifier.deliver_forgot_password(user) if user.password_forgotten
  end
end
```

You might have wondered what we're going to do with the `@password_forgotten` variable in the `User` class, and here's the answer. Our `after_save` method in `UserObserver` kicks in when the `User` object is saved and checks whether the variable is `true`. If yes, it asks the `UserNotifier` mailer to deliver the `forgot_password` mail, passing the current user as an attribute. In normal cases, when `@password_forgotten` is `nil`, the observer does nothing.

Modifying the Controller

The last things to do for our password reset functionality are to tie it all together in `AccountController` and to create views for it. First, to make our new observer work, we need to call it in the controller with the `observer` macro. Add the following code to the `AccountController` class in `app/controllers/account_controller.rb`:

```
class AccountController < ActionController::Base
  observer :user_observer
...
```

Next, we need two actions to support the password reset functionality: one to request the reset code to e-mail, and one to do the actual resetting. We'll call them `forgot_password` and `reset_password`. Let's implement these actions at the end of `AccountController`:

```
def forgot_password
  return unless request.post?
  if @user = User.find_by_email(params[:email])
    @user.forgot_password
    @user.save
    flash[:notice] = "An email with instructions for resetting your password
                      has been sent to your email address."
    redirect_back_or_default(:controller => "/account")
  else
    flash.now[:notice] = "Could not find a user with the given email address."
  end
end

def reset_password
  @page_title = "Reset Password"
  @user = User.find_by_pw_reset_code(params[:id]) rescue nil
  unless @user
    render(:text => "Not found", :status => 404)
    return
  end
  return unless request.post?
  if @user.update_attributes(params[:user])
    @user.reset_password
    flash[:notice] = "Password successfully reset."
    redirect_back_or_default(:controller => "/account")
  end
end
```

`forgot_password` will show a form where the users can add their e-mail address. When the form is submitted to the same action, it will fetch the user with the given e-mail address from the database. Then the action calls the `forgot_password` method for the `@user` object to generate the reset code. Finally, it saves the object, thus triggering the `after_save` call in `UserObserver` and causing the e-mail message with the reset URL to be sent. If everything goes smoothly, the action redirects the user back to where he started the new password request. Otherwise, we'll redisplay the form with an error message.

`reset_password` is the action where George lands when following the link in the e-mail message. It gets the user object by the reset code part of the URL. If the user isn't found, a simple 404 Not Found page is shown. Otherwise, the action shows a form where George can give and confirm a new password. We use postback here as well, meaning that the form is posted to the same `reset_password` action. When the action is run from a `POST` request, we update the password, set the reset code to `nil` calling the `reset_password` method for `@user`, and finally redirect the browser back to wherever George happened to be before trying to log in.

Creating the Form Templates

We need form templates for both the request and password resetting actions. Let's start by creating `app/views/account/forgot_password.rhtml` and adding the following code to it:

```
<p>Give your email address and we'll send you instructions on how to
create a new one.</p>
<%= form_tag %>
<label for="email">Email</label><br />
<%= text_field_tag "email" %><br />
<%= submit_tag "Submit" %>
<%= end_form_tag %>
```

The form is extremely simple, consisting of one text field for the e-mail address and a Submit button, as shown in Figure 8-4.

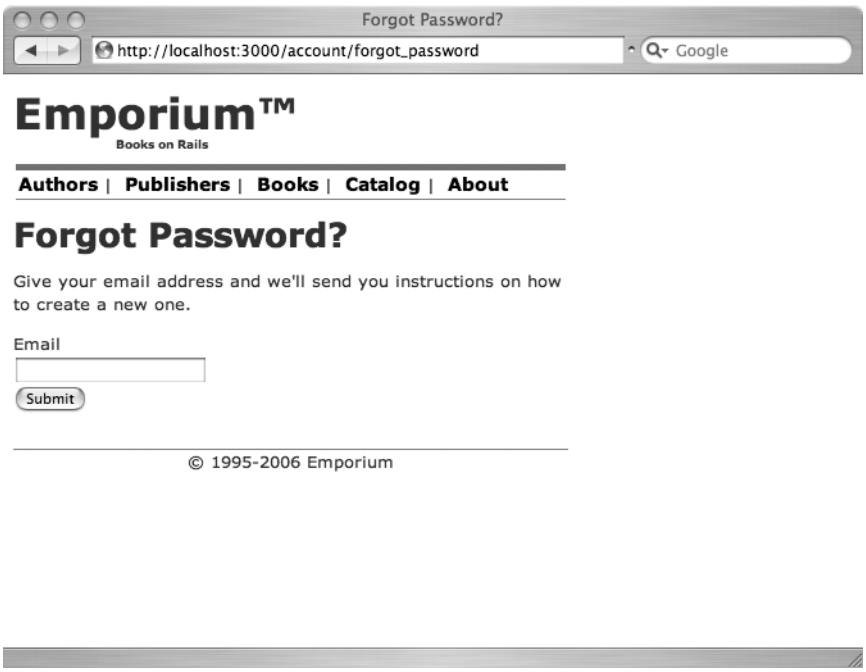


Figure 8-4. *Forgot password? form*

Notice that since we're posting the form back to the current action, we don't even need to specify an address for `form_tag`.

The reset form in `app/views/account/reset_password.rhtml` is almost as simple as the request form:

```
<%= error_messages_for :user %>
<%= form_tag %>
<p><label for="user_password">Password:</label><br />
<%= password_field :user, :password %></p>
<p><label for="user_password_confirmation">Confirm password:</label><br />
<%= password_field :user, :password_confirmation %></p>
<p><%= submit_tag "Submit" %></p>
<%= end_form_tag %>
```

Here, we just show two password fields: one for the actual password and one for a confirmation. Since the `User` class has a `validates_confirmation_of` validation specified for the `password` attribute, the password confirmation is automatically checked against the password. After that, it is stripped from the new `User` object before saving. If the two passwords don't match, `@user` can't be saved, and the form is shown with an error notification by using the `error_messages_for` call, as shown in Figure 8-5.

Figure 8-5. *Error message when passwords do not match*

We now have a working authentication system in our application. It could be easily extended to support open user registration, role-based authentication, reversible encrypted passwords, and "remember me" functionality. For instructions on how to implement these

functions, refer to the plugin's homepage at `http://technoweenie.stikipad.com/plugins/show/Acts+as+Authenticated`.

■ **Note** In the implementation described in this chapter, the current password of a user is not reversible. When a user forgets her password, she must create a new one. The system will not mail her the old one.

Protecting Your Application

Web applications are vulnerable to many exploits, and no framework can make up for a sloppy developer building an application that is easy to hack. In this section, we will review some of the most common exploits and show you how to use Rails to protect your application against them.

Cross-Site Scripting

If you let your users provide content on the site, you must consider that someone may try to enter some malicious content, often in form of JavaScript. Therefore, you should never output anything generated by users directly in the browser. Rails has a shortcut method `h` (alias for `html_escape`), which escapes all the output run through it:

```
<%= h @user.first_name %>
```

For example, if `first_name` is `>George<`, the output of `h` will be `>George<`. That way, a user cannot enter HTML tags or JavaScript and get it parsed by the browser.

If you want to allow the user to store some safe HTML, you can also run the output through the `sanitize` helper, which strips all form tags, script tags, and `onXXX` (such as `onclick`) attributes from tags to prevent running arbitrary JavaScript on the page.

URL and Form Manipulation

It's easy for people to build their own form by copying, for example, your registration form, and adding some fields to it, like this:

```
<input type="hidden" name="user[accepted]" value="1" />
<input type="hidden" name="user[admin]" value="1" />
```

Now suppose the malicious user submits this form to your standard registration action, which has something like this in it:

```
@user = User.create(params[:user])
```

This way, he might end up being an admin user.

It is fairly easy to protect against this type of manipulation in Rails. Just define sensitive attributes as protected:

```
class User < Activerecord::Base
  attr_protected :accepted, :admin
end
```

Now these variables cannot be mass-assigned with a parameters hash like the one in the preceding example. However, you can still set them individually when needed:

```
@user.accepted = true
```

Another vulnerability raises its ugly head when you let users edit their information; for example, with URLs like `http://www.domain.com/posts/edit/28`. It won't take long before someone notices that by changing the last part of the URL, she can get access to posts created by other people. You can protect your application from this by always getting to the objects through the logged-in user:

```
# BAD
@post = Post.find(params[:id])
# GOOD
@post = current_user.posts.find(params[:id])
```

If someone now tries to access a post that doesn't belong to her, an `ActiveRecord::RecordNotFound` exception is raised, which you can then rescue and show a Not Found page to the user:

```
rescue ActiveRecord::RecordNotFound
  return render(:template => "/shared/404", :status => 404)
end
```

SQL Injection

One of the most common security holes in web applications is that they pass user input directly to the database without quoting. Thus, a malicious user can fairly easily run all the SQL he wants to on the server. An example of this would be a search form submission that is handled by the following code:

```
@users = User.find(:conditions => "name = '#{params[:q]}'")
```

Now let's say Dirty Harry puts the following string into the search form:

```
"monkey'; delete from users; --"
```

The resulting SQL query will be as follows:

```
SELECT * from users where name = 'monkey'; delete from users; --'
```

This is a perfectly valid SQL query and will effectively wipe out the whole `users` table. Thus, you should never, ever, pass anything unquoted to the `:conditions` parameter of ActiveRecord finders. Instead, use the bind variable syntax:

```
@users = User.find(:conditions => ["name = ?", params[:q]])
```

You can pass in as many question mark/variable pairs you need. They will be parsed and quoted in the order they are specified.

Another option in simple cases is to use the magic finders, where the parameter value is automatically quoted, too:

```
@users = User.find_by_name(params[:q])
```

Cross-Site Request Forgery

Cross-site request forgery is an attack where, for example, George is tricked into visiting a page where some code attacks Emporium, a site where he is logged in as an administrator. Let's say that George browses to Dirty Harry's site, dirty-harrys.com, where Harry has the following image tag:

```
<img src="http://emporium.com/admin/give_admin_access_to_user/666" />
```

When George visits the page, his browser will try to load an image from the given URL. It won't find an image, but requesting that address gives administrator access to user 666. Note that even though this example uses the `GET` protocol, restricting the URL to `POST` requests doesn't help, because JavaScript can be used to send `POST` requests.

The only way to protect from these kinds of attacks is to use some kind of transient (for example, session-specific) token, in addition to the session cookie, that will be verified upon form postings. You can use a Rails plugin called Security Extensions (`http://wiki.rubyonrails.com/rails/pages/Security+Extensions+Plugin`) to tackle this problem; see its homepage for details. This defense is also effective against the form manipulation threat described earlier.

Summary

In this chapter, we showed you how to integrate a security plugin into your Rails application and how to extend it to reset forgotten passwords. Using the `acts_as_authenticated` plugin, we added support for user authentication. In implementing the reset password functionality, you saw how to use an ActionMailer mailer to send e-mail from your Rails application, as well as how observers can follow the life cycle of ActiveRecord objects and act on events like creating, updating, or deleting an object. Finally, we covered some security problems common to web applications and how to protect your Rails application from them.

In the next chapter, we will finish up the process of buying books from Emporium, by implementing checkout functionality and integration with credit card processing services.

CHAPTER 9

Checkout and Order Processing

In this chapter, we'll implement a checkout page and an order-processing system for the Emporium site. This involves integrating with payment gateways, which George uses for handling the transactions.

We'll show you how to integrate with two popular payment gateways: PayPal and Authorize.Net using two separate frameworks: the Active Merchant plugin and the Payment gem. These two libraries have already implemented the toughest part of the integration with the payment gateway. All we need to do is implement the front-end for the user stories and use the libraries to communicate with the payment gateways.

Towards the end of the chapter, we'll explain how to use the Shipping gem to calculate shipping costs. Lastly, we'll briefly discuss how taxes are calculated.

Getting the Checkout and Order-Processing Requirements

For this sprint, we have four user stories to implement:

- *Check out:* Jill, Emporium's beloved customer, has found two new books that she wants to buy, and has placed them in the shopping cart. The next step for her to continue with the order is to go to the checkout page. Here, she can type in her contact information, the shipping address, and credit card information, and then place the order by submitting the information. This initiates the order-processing workflow that involves billing the customer and shipping the books.
- *View orders*: George needs to be able to view the status of all orders, such as processed orders and closed ones. Processed orders are the ones that have been billed to the customer but have not been shipped yet. Closed orders are the ones that George has sent to the customer.
- *View order:* Before George can ship anything anywhere, he must be able to view the details of the order. We will add a page that shows the shipping address and billing information, along with the contact information for the customer.
- *Close order:* After George has shipped the books, he should close the order. We will set this up on the order details page, so George can simply click a button that sets the order status to closed.

Let's start by implementing the Check Out user story.

Implementing the Check Out User Story

Back in Chapter 5, we implemented the shopping cart for Emporium customers. Now we will complete the web shopping experience by adding the crucial last step: check out, including how to integrate with credit card payment gateways.

Creating the Models

We need a place where we can store the order information. We'll use two tables, named `orders` and `order_items`, which are similar to the `carts` and `cart_items` tables we created in Chapter 5.

You could use single-table inheritance to store both the order items and cart items in the same table, but in this case, we want to clearly separate the two entities. For more information on single-table inheritance, see the API documentation for `ActiveRecord::Base` at `http://rubyonrails.org/api/classes/ActiveRecord/Base.html`.

Creating the Order Model

Begin by creating the `Order` model and the associated migration:

```
$ script/generate model Order
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/order.rb
      create  test/unit/order_test.rb
      create  test/fixtures/orders.yml
      exists  db/migrate
      create  db/migrate/011_create_orders.rb
```

Open `db/migrate/011_create_orders.rb` and replace the contents with the following code:

```
class CreateOrders < ActiveRecord::Migration
  def self.up
    create_table :orders do |t|
      # Contact Information
      t.column :email, :string
      t.column :phone_number, :string
      # Shipping Address
      t.column :ship_to_first_name, :string
      t.column :ship_to_last_name, :string
      t.column :ship_to_address, :string
      t.column :ship_to_city, :string
      t.column :ship_to_postal_code, :string
      t.column :ship_to_country, :string
      # Private parts
      t.column :customer_ip, :string
      t.column :status, :string
      t.column :error_message, :string
      t.column :created_at, :timestamp
      t.column :updated_at, :timestamp
    end
  end

  def self.down
    drop_table :orders
  end
end
```

Columns with names that start with `ship_to` map directly to the shipping information section of the form we'll create later in this chapter. The `email` and `phone_number` fields map to the contact information section of the checkout form. We also want to store private data, including the customer's IP address, so that it is possible to track, for example, credit card frauds.

The `status` field is a string that indicates in which of the following states the order currently is: `open`, `processed`, `closed`, or `failed`. The `open` status is used by default. `processed` is the status of an order for which George has charged the customer. Orders are `closed` when George has verified that the payment has been approved, and after he has sent the books to the

customer. If we receive an error message from the payment gateway, or something else fails in the order processing, we store it in the `error_message` field and set the status to `failed`.

■ **Tip** If you want to keep an audit trail of all the changes that have been done to an order, you can use the `acts_as_versioned` plugin, which can be found at `http://ar-versioned.rubyforge.org/`.

As usual, we drop the table when rolling back changes.

Creating the Order_Item Model

The books that have been ordered also must be stored somewhere, which will be in the `order_items` table. Create the model and the migration with the following command:

```
$ script/generate model Order_Item
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/order_item.rb
   identical  test/unit/order_item_test.rb
   identical  test/fixtures/order_items.yml
      exists  db/migrate
      create  db/migrate/012_create_order_items.rb
```

Open `db/migrate/012_create_order_items.rb` and replace the contents with the following code:

```
class CreateOrderItems < ActiveRecord::Migration
  def self.up
    create_table :order_items do |t|
      t.column :book_id, :integer
      t.column :order_id, :integer
      t.column :price, :float
      t.column :amount, :integer
      t.column :created_at, :timestamp
      t.column :updated_at, :timestamp
    end
  end

  def self.down
    drop_table :order_items
  end
end
```

We store the price of the ordered book and the amount of books ordered. We link each order item to a book and an order.

You can now run the migrations by executing the following:

```
$ rake db:migrate
```

Specifying the Associations

Next, we want to set up the link between the orders and ordered items. We can do this by first changing the `Order` model (`app/models/order.rb`) as shown here:

```
class Order < ActiveRecord::Base
  attr_protected :id, :customer_ip, :status, :error_message, ➥
:updated_at, :created_at

  has_many :order_items
  has_many :books, :through => :order_items

  def total
    order_items.inject(0) {|sum, n| n.price * n.amount + sum}
  end
end
```

Next, change `app/models/order_item.rb` as follows:

```
class OrderItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :book
end
```

Note that we protect the `id`, `customer_ip`, `status`, `error_message`, `updated_at`, and `created_at` fields, so that no one can hack the form and assign a value to these fields, which are used only internally. Protecting the fields is done with the `attr_protected` method, which we introduced in Chapter 8.

■ **Note** In our example, we allow MySQL to generate the order `id`. If you want more complex order numbers, you'll need to generate them yourself.

We have also added a method for calculating the total price of the order. To verify that the relationship between the `Order`, `Order_Item`, and `Book` models are set up correctly, let's create a simple unit test. Open `test/unit/order_test.rb` and add the test shown in Listing 9-1 to it.

Listing 9-1. *Unit Test for Model Relationships*

```
  def test_that_we_can_create_a_valid_order
    order = Order.new(
      # Contact Information
      :email => 'abcdef@gmail.com',
      :phone_number => '3498438943843',
      # Shipping Address
      :ship_to_first_name => 'Hallon',
      :ship_to_last_name => 'Saft',
      :ship_to_address => 'Street',
      :ship_to_city => 'City',
      :ship_to_postal_code => 'Code',
      :ship_to_country => 'Iceland',
      # Billing Information
      :card_type => 'Visa',
      :card_number => '4007000000027',
      :card_expiration_month => '1',
      :card_expiration_year => '2009',
      :card_verification_value => '333'
    )
    # Private parts
    order.customer_ip = '10.0.0.1'
    order.status = 'processed'

    order.order_items << OrderItem.new(
      :book_id => 1,
      :price => 100.666,
      :amount => 13
    )

    assert order.save

    order.reload

    assert_equal 1, order.order_items.size
    assert_equal 100.666, order.order_items[0].price
  end
```

Note that we can't set the `customer_ip` and `status` fields in the same way as the other fields, because they are now protected fields.

Before running the test, remember to copy the changes from the development database to the test by executing the following command:

```
$ rake db:test:clone_structure
```

If you run the test now, it should generate errors related to the billing information fields.

Adding Validations to the Model

We need to validate the billing information, but there's one problem. We don't want to store that information in the database, so there are no fields in our model that we can validate. We can fix this by adding the code we have highlighted here to `app/models/order.rb`:

```
class Order < ActiveRecord::Base
  attr_protected :id, :customer_ip, :status, :error_message, ➥
:updated_at, :created_at
  attr_accessor :card_type, :card_number, ➥
:card_expiration_month, :card_expiration_year, :card_verification_value

  has_many :order_items
  has_many :books, :through => :order_items
```

Using `attr_accessor` has the same result as adding a getter and a setter for each of the attributes. However, it can all be done with one line, instead of something like this:

```
def card_type=(type)
  @card_type = type
end

def card_type
  @card_type
end
```

Collecting correct information on the checkout page is important. If the customer misspells her e-mail address or forgets to enter information in one of the required fields, George might not be able to complete the order. To help prevent this, add the following validations to the `Order` model (`app/models/order.rb`):

```
  validates_size_of :order_items, :minimum => 1
  validates_length_of :ship_to_first_name, :in => 2..255
  validates_length_of :ship_to_last_name, :in => 2..255
  validates_length_of :ship_to_address, :in => 2..255
  validates_length_of :ship_to_city, :in => 2..255
  validates_length_of :ship_to_postal_code, :in => 2..255
  validates_length_of :ship_to_country, :in => 2..255

  validates_length_of :phone_number, :in => 7..20
  validates_length_of :customer_ip, :in => 7..15
  validates_format_of :email, :with => /^([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$/i
  validates_inclusion_of :status, :in => %w(open processed closed failed)
```

```
  validates_inclusion_of :card_type, :in => ['Visa', 'MasterCard', 'Discover'], ➥
:on => :create
  validates_length_of :card_number, :in => 13..19, :on => :create
  validates_inclusion_of :card_expiration_month, ➥
:in => %w(1 2 3 4 5 6 7 8 9 10 11 12), :on => :create
  validates_inclusion_of :card_expiration_year, ➥
:in => %w(2006 2007 2008 2009 2010), :on => :create
  validates_length_of :card_verification_value, :in => 3..4, :on => :create
```

We validate all fields, including credit card information. The credit card fields are validated only on `create`, as they don't exist in the database.

We should also take care that the amount and price are correct, so add the `validate` callback method shown here to `order_items.rb`:

```
class OrderItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :book

  def validate
    errors.add(:amount, "should be one or more") ➥
unless amount.nil? || amount > 0
    errors.add(:price, "should be a positive number") ➥
unless price.nil? || price > 0.0
  end
```

This method validates that the customer is ordering at least one book and that the price is a positive number. Let's add the following test to `test/unit/order_test.rb`, which verifies that all fields are validated:

```
  def test_that_validation_works
    order = Order.new
    assert_equal false, order.save
    # An order should have at least one order item
    assert order.errors.on(:order_items)
    assert_equal 15, order.errors.size
    # Contact Information
    assert order.errors.on(:email)
    assert order.errors.on(:phone_number)
    # Shipping Address
    assert order.errors.on(:ship_to_first_name)
    assert order.errors.on(:ship_to_last_name)
    assert order.errors.on(:ship_to_address)
    assert order.errors.on(:ship_to_city)
    assert order.errors.on(:ship_to_postal_code)
    assert order.errors.on(:ship_to_country)
```

```
    # Billing Information
    assert order.errors.on(:card_type)
    assert order.errors.on(:card_number)
    assert order.errors.on(:card_expiration_month)
    assert order.errors.on(:card_expiration_year)
    assert order.errors.on(:card_verification_value)

    assert order.errors.on(:customer_ip)
  end
```

Run the unit test one last time:

```
$ ruby test/unit/order_test.rb
```

You should see the tests pass.

Creating the Controller and Integration Test

The next task is to create the controller for the Checkout user story. In a console, type the following command:

```
$ script/generate controller Checkout index place_order thank_you
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/checkout
      exists  test/functional/
      create  app/controllers/checkout_controller.rb
      create  test/functional/checkout_controller_test.rb
      create  app/helpers/checkout_helper.rb
      create  app/views/checkout/index.rhtml
      create  app/views/checkout/place_order.rhtml
      create  app/views/checkout/thank_you.rhtml
```

Now, let's continue by using TDD to implement the user story. We want to test two scenarios: one where the cart is empty and one where the cart contains books. We are going to call multiple controllers (the `Cart` and `Checkout` controllers) from our tests, so we'll use an integration test instead of a functional test. Create the test with the following command:

```
$ script/generate integration checkout
```

```
      exists  test/integration/
      create  test/integration/checkout_test.rb
```

Let's start with the first scenario. When the cart is empty, the user shouldn't be able to perform a checkout. This can be verified by replacing the `test_truth` method in `test/integration/checkout_test.rb` with the code shown here:

```
  def test_that_empty_cart_shows_error_message
    get '/checkout'
    assert_response :redirect
    assert_redirected_to :controller => "catalog"
    assert_equal "Your shopping cart is empty! ➥
Please add something to it before proceeding to checkout.", flash[:notice]
  end
```

This test verifies that when the cart is empty, we are redirected to the catalog page and an error message is displayed.

If you run the test now, it fails, because we haven't implemented the controller yet. Open `app/controllers/checkout_controller.rb` in your editor and change the `index` method as follows:

```
  def index
    @order = Order.new
    @page_title = "Checkout"
    if @cart.books.empty?
      flash[:notice] = "Your shopping cart is empty! ➥
Please add something to it before proceeding to checkout."
      redirect_to :controller => 'catalog'
    end
  end
```

Also add the `initialize_cart` filter to the controller:

```
class CheckoutController < ApplicationController
  before_filter :initialize_cart

  def index
```

Recall that this filter was implemented in Chapter 5. It initializes the cart, so that we can access it from the `Checkout` controller and views. It also enables us to show the shopping cart to the right on the page (also implemented in Chapter 5).

Execute the test with the following command:

```
$ ruby test/integration/checkout_test.rb
```

You should see no errors. Perform a manual test by opening `http://localhost:3000/checkout`. Because the shopping cart is empty, you should be redirected to the catalog page, where the error message in Figure 9-1 is displayed.

Figure 9-1. *The catalog page displaying an empty cart error message*

Next, we'll add a test for the second scenario. When the shopping cart contains one or more items (books), we want to display a form with three sections: contact information, shipping address, and billing information. First, add the following fixtures and test to `test/integration/checkout_test.rb`:

```
fixtures :authors, :publishers, :books

def test_that_placing_an_order_works
  post '/cart/add', :id => 1
  get '/checkout'
  assert_response :success
  assert_tag :tag => 'legend', :content => 'Contact Information'
  assert_tag :tag => 'legend', :content => 'Shipping Address'
  assert_tag :tag => 'legend', :content => 'Billing Information'
end
```

The test begins by adding a book (defined in the `books` fixture file) to the cart by calling the `/cart/add` action. Then it accesses the checkout page and verifies that the request is successful. It proceeds by checking that the page contains the three required sections. This is done by looking for three `legend` tags having the following content: `Contact Information`, `Shipping Address`, and `Billing Information`.

Now that we have the new test in place, we could try to run it, but it would fail miserably when it tries to find the three sections.

Creating the View

To build the checkout view, open `app/views/checkout/index.rhtml` and add the following code:

```
<%= error_messages_for 'order' %>
<p><em>Your order is displayed in the shopping cart to the right.</em></p>
<form method="post" id="checkout" action="<%= url_for :action => :place_order %>">
  <fieldset>
    <legend>Contact Information</legend>
    <p>
    <label for="order_email">Email</label><br/>
    <%= text_field :order, :email %>
    </p>
    <p>
    <label for="order_phone_number">Phone number</label><br/>
    <%= text_field :order, :phone_number %>
    </p>
  </fieldset>
  <p>
    <%= submit_tag "Place Order" %>
  </p>
</form>
```

The code contains the checkout form and the contact information section. Notice that we are using the `fieldset` and `legend` tags, which are good from a usability point of view, to group the `email` and `phone_number` fields. We are trying hard not to overwhelm the customers with fields that they need to fill in.

■ **Tip** The checkout page is probably your site's most important page. You definitely don't want a customer to cancel an order on the checkout page. To make the checkout process faster for returning customers, you could ask them to register. This would allow you to save the customer's contact and shipping information, and prepopulate the form on the checkout page.

Next, add the shipping information section (below the </fieldset> tag) with the code shown here:

```
  <fieldset>
    <legend>Shipping Address</legend>
    <p>
    <label for="order_ship_to_first_name">First name</label><br/>
    <%= text_field :order, :ship_to_first_name %>
    </p>
    <p>
    <label for="order_ship_to_last_name">Last name</label><br/>
    <%= text_field :order, :ship_to_last_name %>
    </p>
    <p>
    <label for="order_ship_to_address">Address</label><br/>
    <%= text_field :order, :ship_to_address %>
    </p>
    <p>
    <label for="order_ship_to_city">City</label><br/>
    <%= text_field :order, :ship_to_city %>
    </p>
    <p>
    <label for="order_ship_to_postal_code">Postal/Zip code</label><br/>
    <%= text_field :order, :ship_to_postal_code %>
    </p>
    <p>
    <label for="order_ship_to_country">Country</label><br/>
    <%= country_select(:order, :ship_to_country, ➥
priority_countries = ['United States']) %>
    </p>
  </fieldset>
```

This section contains six fields to collect the customer's first name, last name, address, city, postal code, and country. Note that the country field displays a list of all countries in the world. The list is generated by the built-in Rails method country_select:

```
  country_select(:order, :ship_to_country, priority_countries = ['United States'])
```

This binds the `country` field to the `order` model's `ship_to_country` field, which we'll create later. The `priority_countries` parameter is an array of strings that specifies which countries should be displayed at the top of the list, as shown in Figure 9-2.

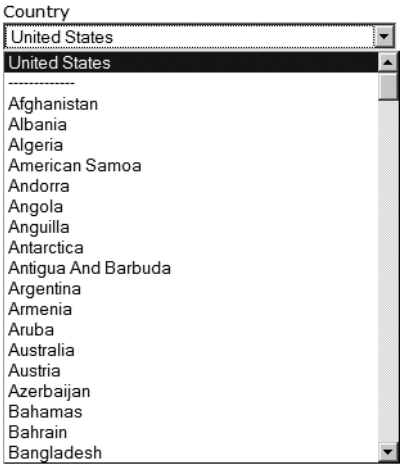


Figure 9-2. *The country selection list*

Next, add the billing information section directly after the last `</fieldset>` tag:

```
  <fieldset>
    <legend>Billing Information</legend>
    <p>
    <label for="order_card_type">Credit card type</label><br/>
    <select name="order[card_type]" id="order_card_type">
      <%= options_for_select([ "Visa", "MasterCard", "Discover" ], ➥
@order.card_type) %>
    </select>
    </p>
    <p>
    <label for="order_card_expiration_month">Expiration date</label><br/>
    <select name="order[card_expiration_month]">
    <%= options_for_select(%w(1 2 3 4 5 6 7 8 9 10 11 12), ➥
 @order.card_expiration_month) %>
    </select>
    /
    <select name="order[card_expiration_year]">
<%= options_for_select(%w(2006 2007 2008 2009 2010 2011), ➥
 @order.card_expiration_year) %>
    </select>
    </p>
```

```
    <p>
      <label for="order_card_number">Card number</label><br/>
      <%= text_field :order, :card_number %>
    </p>
    <p>
      <label for="order_card_verification_value"> ➥
      <abbr title="Card Verification Value">CVV</abbr>/ ➥
<abbr title="Card Validation Check">CVC</abbr>
      </label><br/>
      <%= text_field :order, :card_verification_value %>
    </p>
  </fieldset>
```

This section contains fields for the credit card type, expiration date, card number, and card verification code (CVC).

■ **Note** The card verification code is used for fraud prevention. For more information, see `http://en.wikipedia.org/wiki/Card_Verification_Code`.

The options for drop-down lists, such as the one that lists credit cards, are generated with the `options_for_select` helper method. This method generates one `<option>` tag for each item in the specified array. With the second parameter, we specify the item that should be selected from the list.

Everything required for the test to pass is now in place, so execute it by issuing the following command:

```
$ ruby test/integration/checkout_test.rb
```

This time, all tests should pass. Before we test that it works with our browser, we'll add a checkout link to the shopping cart that we created in Chapter 5. Open `app/views/cart/_cart.rhtml` and add the highlighted code:

```
  <h3>Your Shopping Cart</h3>
  <p>
    <strong>
    <%= link_to "Proceed to Checkout", :controller => 'checkout' ➥
unless controller.controller_name == 'checkout' %>
    </strong>
  </p>
  <ul>
    <% for item in @cart.cart_items %>
    <li id="cart_item_<%= item.book.id %>">
      <%= render :partial => "cart/item", :object => item %>
    </li>
    <% end %>
  </ul>
```

Note that we show the Proceed to Checkout link only if the user is not already on the checkout page.

We want to make the checkout page look a bit nicer, so add the code shown in Listing 9-2 to the style sheet (`app/public/stylesheets/style.css`).

Listing 9-2. *Additions to the Style Sheet*

```
#checkout fieldset {
  border-top: 1px solid #efefef;
  border-left: 1px solid #efefef;
  border-bottom: 1px solid #ccc;
  border-right: 1px solid #ccc;
  padding: 1em 1em 1em 1.5em;
  width: 300px;
  margin-bottom: 10px;
}

#checkout fieldset:hover {
  border: 1px solid #3A789D;
}

#checkout legend {
  font-weight: bold;
}

#checkout fieldset input {
  margin: 1px;
}

#order_card_verification_value {
  width: 50px;
}

#checkout fieldset input:focus {
  background-color: #cccccc;
}
```

Open `http://localhost:3000/catalog` in your browser and add a couple of books to the shopping cart. Then click the Proceed to Checkout link. You should now see the checkout page, as shown in Figure 9-3.

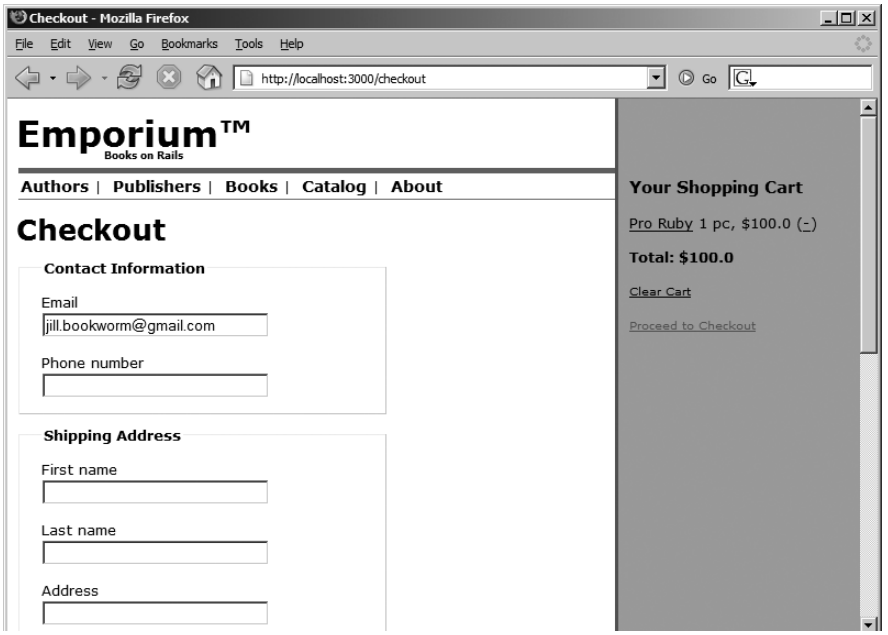


Figure 9-3. *The checkout page*

To the right, the shopping cart is displayed, so that the customer can view the contents of it and remove books before placing the order. At the bottom of the page is a Place Order button, which will just show an exception if you click it now. Two things should happen when this button is clicked:

- The order information, including the amount and price of ordered books, contact information, and shipping address, should be saved to the database.
- The total price of the order should be charged to the credit card specified in the billing information section.

Note that it's the payment gateway that actually charges the credit card, and that the payment gateway is the only system that needs the credit card information. We don't need to store the credit card information in the database.

With the controller and view in place, we can continue. Let's first implement the part that stores the data in the database. Then we'll tackle the integration with the payment gateway.

STORING CREDIT CARD INFORMATION

You should be very careful when handling credit card information. If your business doesn't depend on it, never store the information on your system. In case you need it, for instance, when implementing a checkout process similar to Amazon's One-Click Shopping (`http://cse.stanford.edu/class/cs201/projects-99-00/software-patents/amazon.html`), you should read about the Cardholder Information Security Program (CISP) and Payment Card Industry (PCI) security standards, which mandate that you never under any circumstances store the CVC2/CVV2/CID, PIN, or magnetic stripe data on your system.

For more information about the security standards, see the following:

- `http://usa.visa.com/business/accepting_visa/ops_risk_management/cisp.html`
- `www.pcisecuritystandards.org/tech/download_the_pci_dss.htm`

James Duncan Davidson has also written a good summary of the issues related to handling credit card information. You can read it at `http://blog.duncandavidson.com/2006/06/cautious_advice.html`.

Saving the Order Information

We are now confident that the checkout form and the two new ActiveRecord models work as we intended, so let's continue with the next part of the checkout page implementation: the `place_order` action. This action is triggered when the user has filled out the form and clicks the Place Order button.

Updating the Integration Test

First, we change the integration test (`test/integration/checkout_test.rb`) we created earlier as shown here:

```
def test_placing_order
  post '/cart/add', :id => 1
  get '/checkout'
  assert_response :success
  assert_tag :tag => 'legend', :content => 'Contact Information'
  assert_tag :tag => 'legend', :content => 'Shipping Address'
  assert_tag :tag => 'legend', :content => 'Billing Information'

  post '/checkout/place_order', :order => {
    # Contact Information
    :email => 'abce@gmail.com',
    :phone_number => '3498438943843',
    # Shipping Address
    :ship_to_first_name => 'Hallon',
    :ship_to_last_name => 'Saft',
    :ship_to_address => 'Street',
    :ship_to_city => 'City',
```

```
      :ship_to_postal_code => 'Code',
      :ship_to_country => 'Iceland',
      # Billing Information
      :card_type => 'Visa',
      :card_number => '4007000000027',
      :card_expiration_month => '1',
      :card_expiration_year => '2009',
      :card_verification_value => '333',
    }

    assert_response :redirect
    assert_redirected_to '/checkout/thank_you'
  end
```

This will place an order for one book and check that the order process was successful, which is indicated by a redirect to the page where we show a thank you message to the customer.

Adding the place_order Action

Next, we add the `place_order` action to the `checkout` controller (`app/controllers/checkout_controller.rb`):

```
  def place_order
    @page_title = "Checkout"
    @order = Order.new(params[:order])
    @order.customer_ip = request.remote_ip
    populate_order

    if @order.save
      if @order.process
        flash[:notice] = 'Your order has been submitted, ➥
and will be processed immediately.'
        session[:order_id] = @order.id
        # Empty the cart
        @cart.cart_items.destroy_all
        redirect_to :action => 'thank_you'
      else
        flash[:notice] = "Error while placing order. ➥
'#{@order.error_message}'"
        render :action => 'index'
      end
    else
      render :action => 'index'
    end
  end
```

The `place_order` action calls two methods, `populate_order` and `order.process`, which we haven't created yet. The `populate_order` method (added in `checkout_controller.rb`) simply copies the books from the shopping cart to the order:

```
private

def populate_order
  for cart_item in @cart.cart_items
    order_item = OrderItem.new(
      :book_id => cart_item.book_id,
      :price => cart_item.price,
      :amount => cart_item.amount
    )
    @order.order_items << order_item
  end
end
```

The following code for the `process` method should be added to the `Order` model (`app/models/order.rb`):

```
private

def process
  result = true
  #
  # TODO Charge the customer by calling the payment gateway
  #
  self.status = 'processed'
  save!
  result
end
```

`process` is where we charge the customer for the total amount of the order, and this is where we'll put the payment gateway integration code.

You can run the integration test by executing the following command:

```
$ ruby test/integration/checkout_test.rb
```

Although we haven't implemented the thank you page yet, the test will pass. Open `app/views/checkout/thank_you.rhtml` and add the following code to it:

```
For future references use invoice number <%= session[:order_id] %>
```

Next, change the `thank_you` action in the `Checkout` controller (`app/controllers/checkout_controller.rb`) as follows:

```
def thank_you
  @page_title = 'Thank You!'
end
```

To test the thank you page, add a couple of books to the shopping cart and click the checkout link. Then fill out the checkout form and place the order. You should now see the thank you page, as shown in Figure 9-4.

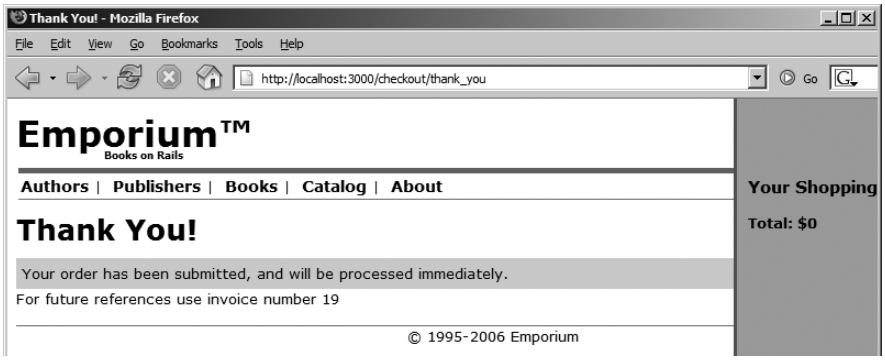


Figure 9-4. *The thank you page*

Integrating with Payment Gateways

You can think of a payment gateway as a kind of proxy that handles the tricky details of credit card transactions for merchants. You send the payment details to the gateway, and the gateway forwards the details to the financial institution. The payment is then processed (with a possible delay) by the financial institution, after which the result is communicated back to the gateway, and then to you or your application.

We'll first show you how to use Active Merchant, a payment abstraction library, to integrate with PayPal, the leading online payment solution that enables merchants to manage credit card transactions online. Later, we'll also demonstrate how to integrate with Authorize.Net, another leading payment gateway, including how to do this with the Payment gem as an alternative to Active Merchant.

■ **Note** Only US-based businesses are allowed to use PayPal's Website Payments Pro and Authorize.Net. For alternative payment gateways, see the Active Merchant homepage at `http://home.leetsoft.com/am`. And remember, using payment gateways is not the only option. Some banks allow you to add a simple form to your site, which the customer can use for transferring money directly to your bank account.

Installing the Active Merchant Plugin

Active Merchant (`http://home.leetsoft.com/am`) is a Rails plugin, written by Tobias Lütke and various contributors, that allows you to integrate with various payment gateways, including: Moneris, Authorize.Net, TrustCommerce, PsiGate, and PayPal's Website Payments Pro. It's currently used by Shopify (`http://shopify.com`) in production.

You first need to install the Money gem (`http://rubyforge.org/projects/money/`), which Active Merchant uses to handle money:

```
$ sudo gem install money
```

Next, install Active Merchant itself with the following command:

```
$ script/plugin install svn://home.leetsoft.com/active_merchant/trunk/ ➥
active_merchant
```

The command checks out the latest version of the plugin and places it in the `vendor/plugins/active_merchant` directory.

Active Merchant uses the SOAP protocol to communicate with PayPal. This means you'll need to install `soap4r`, an implementation of SOAP 1.1. Download the latest version from `http://dev.ctor.org/soap4r`, extract the package to a folder of your choice, and then execute the following command:

```
$ sudo ruby install.rb
```

■ **Note** At the time of writing, Active Merchant required `soap4r` version 1.5.5 or greater to work. To retrieve the latest version, execute `svn checkout http://dev.ctor.org/svn/soap4r/trunk soap4r`.

`soap4r` uses the `http-access2` library to communicate with the PayPal servers, so download the latest stable release of this library from `http://dev.ctor.org/http-access2/`, extract the package, and install it with the following command:

```
$ sudo ruby install.rb
```

That's it. If you have the server running, restart it for the changes to take effect.

Integrating with PayPal

To be able to test PayPal, you will first need to sign up for an account at PayPal Developer Central (`https://developer.paypal.com/`). After you log in, you are greeted with the page shown in Figure 9-5.

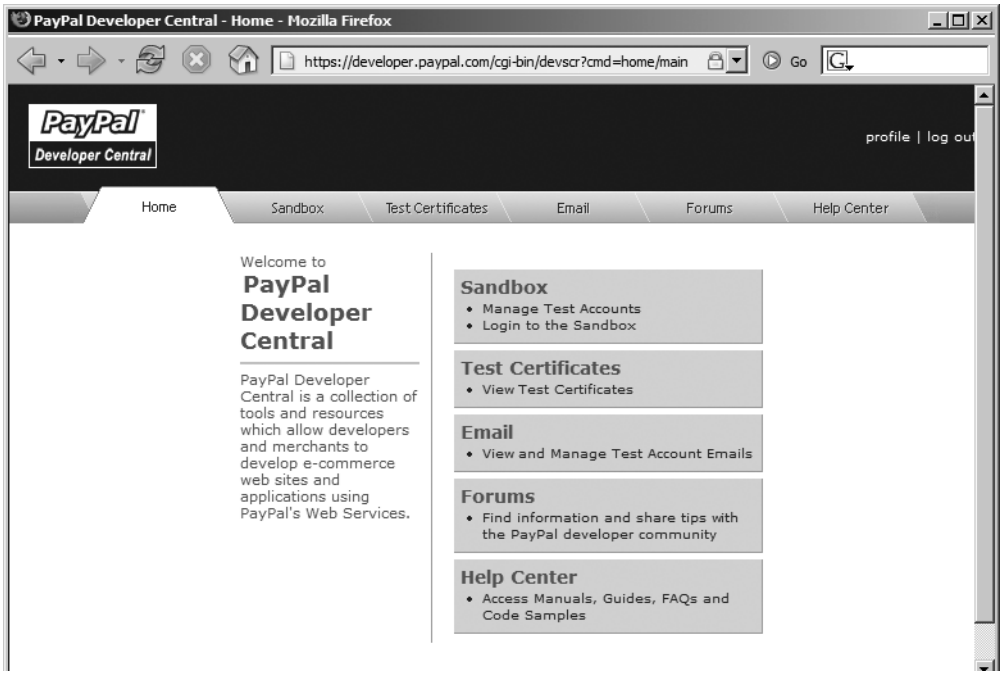


Figure 9-5. *The PayPal Developer Central homepage*

Along with links to the forums and help references, PayPal Developer Central provides access to three important areas:

- *Sandbox:* This is where you can create dummy bank accounts and credit cards that you can use for testing transactions without actually billing anyone.
- *Test Certificates*: This is where you can download the test certificate, which you'll need when communicating with PayPal over a secure SSL connection.
- *Email*: This page is where all e-mail messages that PayPal sends end up. Instead of sending them to your real account, PayPal simply stores them on its servers and displays them on this page.

Using PayPal Developer Central, we will first create a business account and credit card in the Sandbox. This will include steps for verifying and confirming the account. Then we will set up the API credentials that Active Merchant needs when communicating with PayPal, by creating a private key and certificate. Finally, we can return to our application and implement the integration with the payment gateway.

Creating a Dummy Bank Account and Credit Card

The Sandbox is a safe testing environment and replica of the production PayPal environment. We'll use it to create a business account and a dummy credit card for our tests.

From the PayPal Developer Center homepage, click the Sandbox link to open the PayPal Sandbox page. Click the Create Account link and select Business Account. To create the business account, follow the suggested steps. Remember to write down the credit card information, because you'll need it later.

Once you have created the test account, you should see it listed on the Sandbox page, as shown in Figure 9-6.

Figure 9-6. *The PayPal Sandbox page displaying the test account*

Continue by clicking the Launch Sandbox button beneath the Test Accounts listing. This opens the Sandbox login page. Log in, and you should see the business account, as shown in Figure 9-7.

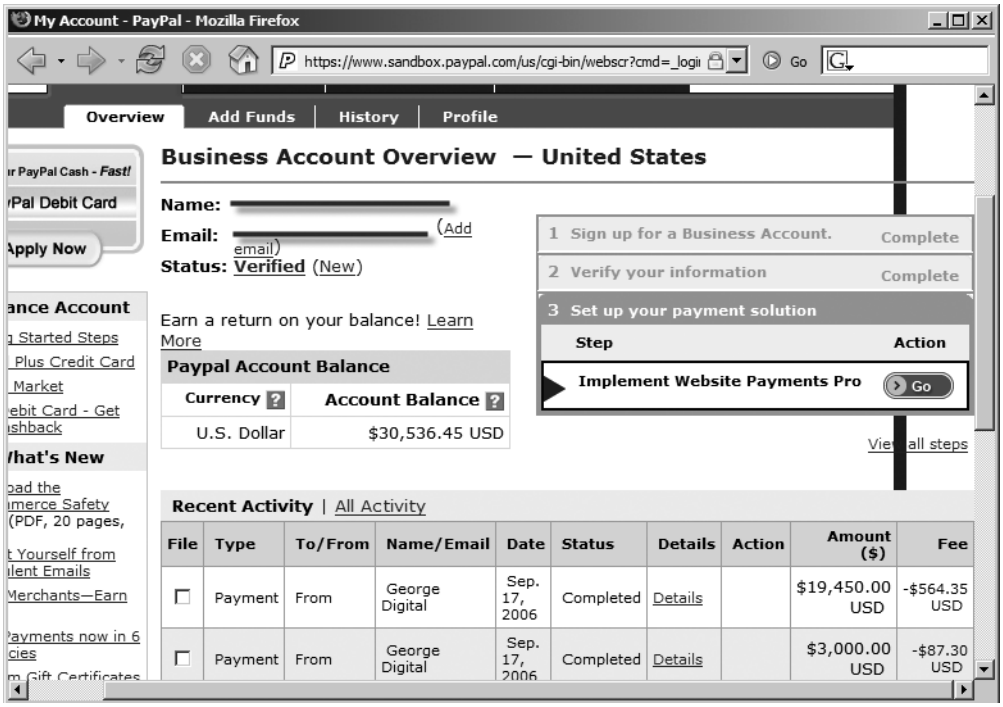


Figure 9-7. *The business account displayed in the PayPal Sandbox*

Here, you can see that the status of the account is verified. In the example in Figure 9-7, we have already made some test transactions, which is why the account balance is $30,536,45.

Creating API Credentials

Next, we'll create the credentials that Active Merchant needs when communicating with PayPal. Follow these steps:

1. From the business account display (Figure 9-7), click the Profile tab. Then click Request API Credentials.

2. On the next page, select API SSL Client-Side Certificate, agree to the terms by selecting the appropriate option, and click the Submit button.

3. On the following page, you should see the API username and API password, as shown in Figure 9-8. Click the Download Certificate button. This will download a file (`cert_key_pem.txt`) containing the private key and the certificate, in the following format:

```
-----BEGIN RSA PRIVATE KEY-----
...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----
```

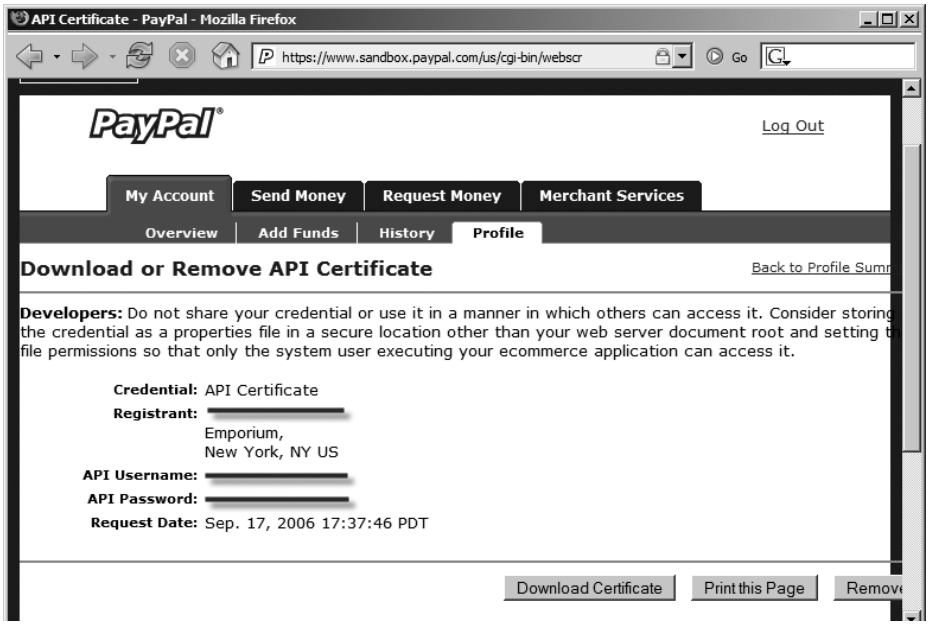


Figure 9-8. *The Download or Remove API Certificate page*

4. Select and copy the private key, including the start and end tags, and save it in `config/paypal/sandbox.key`.

5. Select and copy the certificate and save it in `config/paypal/sandbox.crt`.

6. Download one of the PayPal Software Development Kits (SDKs), such as the PHP SDK, from `http://developer.paypal.com`. Then copy the server certificate (`api_cert_chain.crt`) to `config/paypal/api_cert_chain.crt`.

This completes the PayPal signup and account configuration process. Now we're ready to use Active Merchant and the PayPal account.

Setting Up PayPal Transactions

We can now start using the test account. First, change the `process` method in the `Order` model (`app/models/order.rb`) as shown here:

```
def process
  if closed? raise "Order is closed"
  begin
    process_with_active_merchant
  rescue => e
    logger.error("Order #{id} failed with error message #{e}")
    self.error_message = 'Error while processing order'
    self.status = 'failed'
  end
  save!
  self.status == 'processed'
end
```

We'll call the payment gateway from the `process` method. If there's an exception in the order-processing code, we set the order status to `failed` and log the error message to the standard log. The main logic is located in the `process_with_active_merchant` method, which should be added to `app/models/order.rb`:

```
def process_with_active_merchant
  Base.gateway_mode = :test

  gateway = PaypalGateway.new(
    :login     => 'business_account_login',
    :password  => 'business_account_password',
    :cert_path => File.join(File.dirname(__FILE__), "../../config/paypal")
  )
  gateway.connection.wiredump_dev = STDERR

  creditcard = CreditCard.new(
    :type                => card_type,
    :number              => card_number,
    :verification_value  => card_verification_value,
    :month               => card_expiration_month,
    :year                => card_expiration_year,
    :first_name          => ship_to_first_name,
    :last_name           => ship_to_last_name
  )
```

```
    # Buyer information
    params = {
      :order_id => self.id,
      :email => email,
      :address => { :address1 => ship_to_address,
                    :city => ship_to_city,
                    :country => ship_to_country,
                    :zip => ship_to_postal_code
                  } ,
      :description => 'Books',
      :ip => customer_ip
    }

    response = gateway.purchase(total, creditcard, params)

    if response.success?
      self.status = 'processed'
    else
      self.error_message = response.message
      self.status = 'failed'
    end
  end
```

We are setting the gateway mode to `test`, because we want to use the Sandbox instead of the live environment.

We set the `wiredump_dev` parameter to true by using `gateway.connection.wiredump_dev = STDERR`. This prints out the HTTP traffic to the console, which helps debug the traffic between the server and the gateway.

Note that PayPal expects the `country` field to contain the country code, not the country name, which is the case at the moment. We can fix this by hard-coding the list of countries and codes in the view (`app/views/checkout/index.rhtml`), as shown here:

```
    <select name="order[ship_to_country]">
      <option value="FI">Finland</option>
      <option value="NO">Norway</option>
      <option value="SE">Sweden</option>
      <option value="DK">Denmark</option>
    </select>
```

■ **Note** Another option for getting the country codes is to use the TZInfo library (`http://tzinfo.rubyforge.org/`), as explained at `http://rails.techno-weenie.net/tip/2006/6/5/country_select_with_country_codes`.

We can now test that transactions are sent to PayPal, by adding a few books to the shopping cart and placing an order. Then log in to the PayPal Sandbox (`www.sandbox.paypal.com/`) with the dummy account you created earlier. You should see the order on the History page, as shown in Figure 9-9.

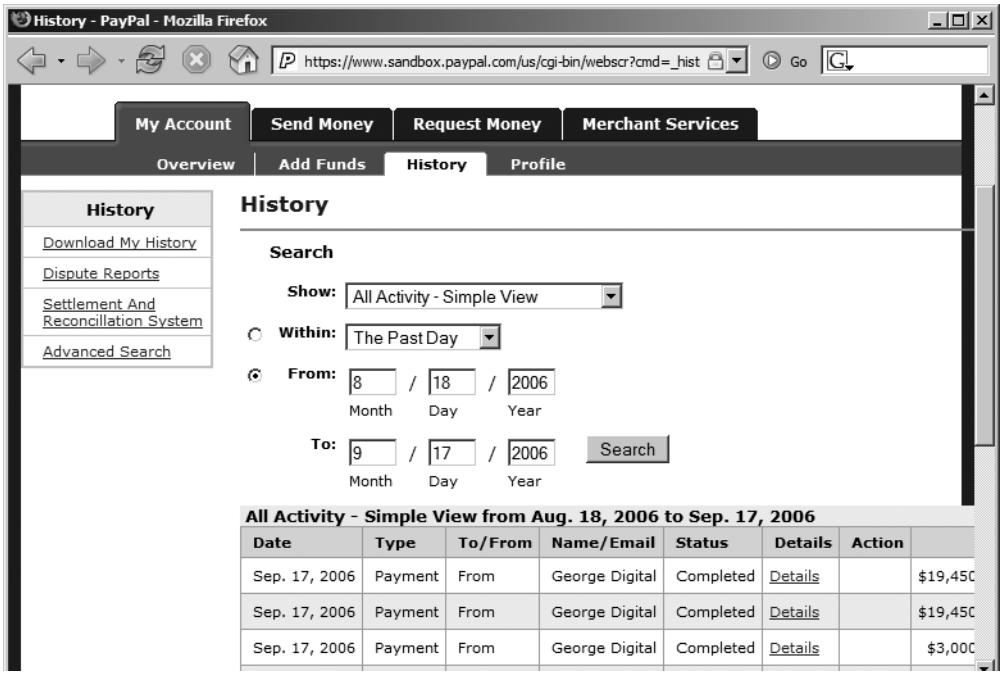


Figure 9-9. *The PayPal History page displaying transactions*

Note that the signup and verification e-mail messages are not sent to the e-mail address you specify on the signup forms. To view the messages, go to the Email page in PayPal Developer Central, shown in Figure 9-10, which is accessible after you log in.

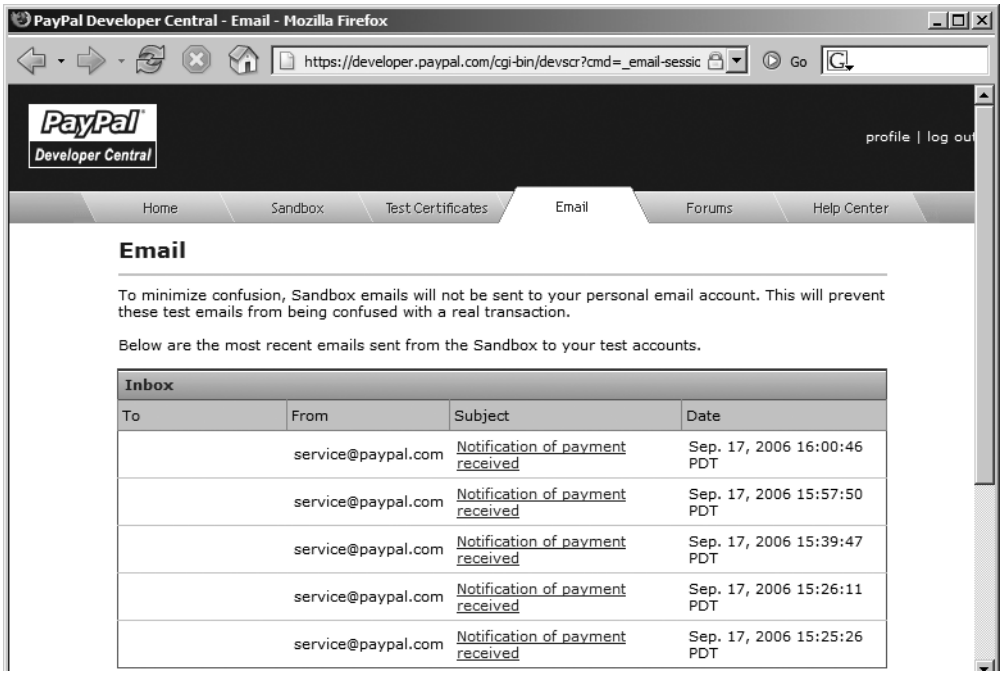


Figure 9-10. *The Email page displaying payment notifications*

Integrating with Authorize.Net

Authorize.Net is an alternative to PayPal. To be able to test Authorize.Net, you'll first need to apply for a test account. This account works in the same way as a real account, except that no one is billed. This means you can play around with the code, without the fear of losing money.

To apply for a test account, go to the `http://developer.authorize.net/testaccount/` page and fill out the form. You should receive an e-mail message with the test account information.

■ **Note** For more information about the Authorize.Net payment integration API that these libraries implement, refer to the Advanced Integration Method (AIM) documentation at `http://www.authorize.net/support/AIM_guide.pdf`.

We are now ready to use Active Merchant for billing the customer. The first step is to include Active Merchant in the `Order` model, by adding the following code to the first line in `app/models/order.rb`:

```
include ActiveMerchant::Billing
```

The code for the Active Merchant version of the `process` method (`process_with_active_merchant`) is shown in Listing 9-3. Add it to `app/models/order.rb`.

Listing 9-3. *The process_with_active_merchant Method*

```
def process_with_active_merchant
  creditcard = ActiveMerchant::Billing::CreditCard.new({
    :type => card_type,
    :number => card_number,
    :month => card_expiration_month,
    :year => card_expiration_year,
    :first_name => ship_to_first_name,
    :last_name => ship_to_last_name
  })

  if creditcard.valid?
    gateway = AuthorizedNetGateway.new({
      :login => "your login",
      :password => "your password"
    })
    options = {
      :card_code => card_verification_value,
      :name => ship_to_first_name + " " + ship_to_last_name,
      :address => ship_to_address,
      :city => ship_to_city,
      :zip => ship_to_postal_code,
      :country => ship_to_country,

      :email => email,
      :phone => phone_number,
      :customer_ip => customer_ip
    }
    response = gateway.purchase(total, creditcard, options)

    if response.success?
      self.status = 'processed'
    else
      self.status = 'failed'
      self.error_message = response.message
    end
  else
    self.status = 'failed'
    self.error_message = 'Invalid credit card'
  end
end
```

We first create a credit card object and check that it is valid, using the Active Merchant API. Then we call the `purchase` method on the gateway, passing in the total amount of the order, the credit card information, and the options, including contact information and shipping address.

Lastly, we check the response from the gateway, and set the status to `processed`. If the gateway returns an error, we set the status to `failed` and store the error message in the `error_message` field.

To see if it works, add some books to the shopping cart and check out. Then go to `https://test.authorize.net/` and log in with the credentials you received from Authorize.Net. You should see the transaction listed on the Unsettled Transactions page, as shown in Figure 9-11.

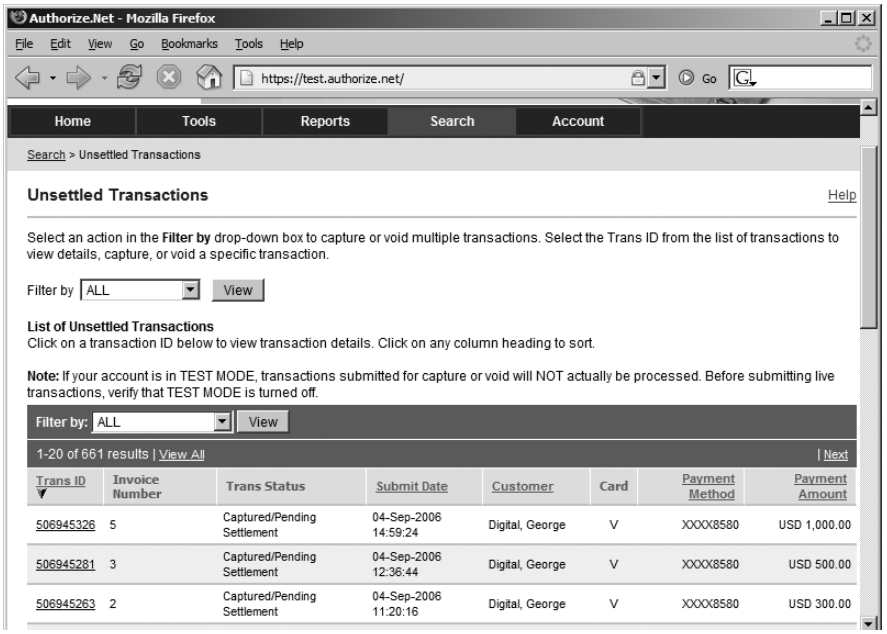


Figure 9-11. *The Authorize.Net Unsettled Transactions page*

Click the transaction ID shown in the first column on the left side of the Unsettled Transactions page. You will see the Transaction Detail page, as shown in Figure 9-12. On this page, you should see the contact information and shipping information that the user entered on your site.

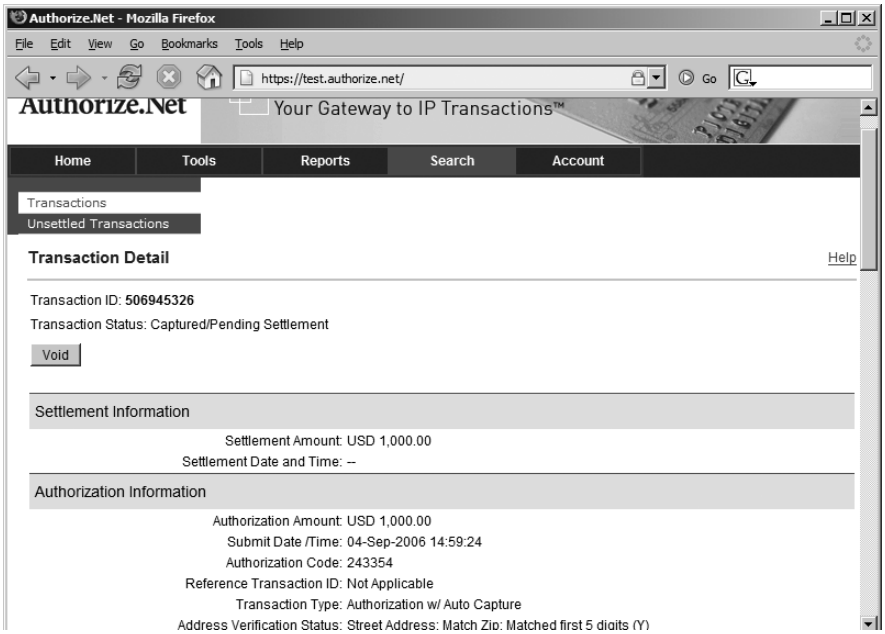


Figure 9-12. *Authorize.Net Transaction Detail page*

Later, when the funds are transferred by the financial institution from the customer to your bank account, you can find the transaction listed on the Authorize.Net Transactions page.

Using the Payment Gem

The Payment gem (developed by Lucas Carlson) is an alternative to Active Merchant that can also be used to integrate with Authorize.Net. It is a bit easier to use than Active Merchant, but it doesn't support multiple gateways. The project is also not being as actively developed as Active Merchant; the last release was in June 2005.

Downloading and installing the Payment gem is easy. Simply execute the following command:

```
$ sudo gem install payment
```

Before you can use the Payment gem, you need to create a configuration file that holds the login name and password to Authorize.Net. Save the following in `config/payment.yml`:

```
username: <Your login>
transaction_key: <Your password>
```

Next, add the `require` line to the `Order` model (`app/models/order.rb`):

```
require 'payment/authorize_net'
include ActiveMerchant::Billing

class Order < ActiveRecord::Base
```

Then change the `process` method as follows:

```
def process
  begin
    process_with_payment_gem
  rescue => e
    logger.error("Order #{id} failed with error message #{e}")
    self.error_message = 'Error while processing order'
    self.status = 'failed'
  end
  save!
  self.status == 'processed'
end
```

We changed only one line, which allows us to easily switch back to use Active Merchant. As usual, the code for the `process_with_payment_gem` method, shown in Listing 9-4, should be added to `app/models/order.rb`.

Listing 9-4. *The process_with_payment_gem Method*

```
def process_with_payment_gem
    transaction = Payment::AuthorizeNet.new(
      :prefs       => "#{RAILS_ROOT}/config/payment.yml",
      :login => 'your login',
      :password => 'your password',
      :url => 'https://test.authorize.net/gateway/transact.dll',
      :amount => total,
      :card_number => card_number,
      :expiration  => "#{card_expiration_month}/#{card_expiration_year}",
      :first_name  => ship_to_first_name,
      :last_name   => ship_to_last_name,
      :ship_to_last_name => ship_to_last_name,
      :ship_to_first_name => ship_to_first_name,
      :ship_to_address => ship_to_address,
      :ship_to_city => ship_to_city,
      :ship_to_zip => ship_to_postal_code,
      :ship_to_country => ship_to_country,
      :customer_ip => customer_ip,
      :invoice_num => id
     )
    begin
      transaction.submit
      logger.debug(
        "Card processed successfully.
          Response codes:
            authorization: #{transaction.authorization}
            result code: #{transaction.result_code}
            avs code: #{transaction.avs_code}
            transaction id: #{transaction.transaction_id}
            md5: #{transaction.md5}
            cvv2 response: #{transaction.cvv2_response}
            cavv response: #{transaction.cavv_response}"
      )
      self.status = 'processed'
    rescue => e
      self.error_message = transaction.error_message
      self.status = 'failed'
    end
  end
```

Note that we are specifying the test URL that we received from Authorize.Net here.

We now have all code in place, so perform a manual test by shopping for some books and placing the order on the checkout page. The result should be the same as with Active Merchant: you should see the order on the Authorize.Net Unsettled Transactions page.

Implementing the Administrator User Stories

Next, we'll build the administrator interface for managing orders. After the system has sent the payment request to the payment gateway, George needs to log in to the payment gateway and verify that the transaction has been settled, meaning that the funds have been transferred from the customer to George's bank account. There might be a delay between the request and when the funds are actually transferred.

When George goes to the order administration interface, he wants to see a list of all orders, sorted by the date they were created. He also wants to list orders by status, so we'll create a view for listing all orders, as well as individual views for listing orders by their status: `open`, `processed`, `closed`, and `failed`.

Implementing the View Orders User Story

The first step in implementing this View Orders user story is to generate a controller:

```
$ script/generate controller 'admin/order' index show close
```

```
      exists  app/controllers/admin
      exists  app/helpers/admin
      exists  app/views/admin/order
      exists  test/functional/admin
      create  app/controllers/admin/order_controller.rb
      create  test/functional/admin/order_controller_test.rb
      create  app/helpers/admin/order_helper.rb
      create  app/views/admin/order/index.rhtml
      create  app/views/admin/order/show.rhtml
      create  app/views/admin/order/close.rhtml
```

Open app/controllers/admin/order_controller.rb and replace the empty index method with the code shown here:

```
class Admin::OrderController < ApplicationController

  def index
    @status = params[:id]
    if @status.blank?
      @status = 'all'
      conditions = nil
    else
      conditions = "status = '#{@status}'"
    end

    @page_title = "Listing #{@status} orders"
    @order_pages, @orders = paginate :orders, :per_page => 10, ➥
:order => 'created_at desc', :conditions => conditions
  end
```

First, we retrieve the status parameter, which we'll use to filter the list of orders. If it is blank, we simply show all orders. If the status parameter is not blank, then we use it to build the conditions parameter for the paginate method. We also set the order parameter so that the list is sorted.

Next, we'll create a menu that allows George to filter the list of orders easily. Save the code shown here in app/views/admin/order/_navigation.rhtml:

```
<p>
  View: <%= link_to "all", :id => '' %>,
  <%= link_to "open", :id => 'open' %>,
  <%= link_to "processed", :id => 'processed' %>,
  <%= link_to "closed", :id => 'closed' %>,
  <%= link_to "failed", :id => 'failed' %>
</p>
```

We'll later use the same partial on the order details page, but for now, we'll just use it on the view orders page (`app/views/admin/order/index.rhtml`), which is shown here:

```
<%= render :partial => 'navigation' %>
<table>
  <tr>
    <th>ID</th>
    <th>Status</th>
    <th>Total amount</th>
    <th>Size</th>
    <th>Created at</th>
    <th>Updated at</th>
    <th></th>
  </tr>
<% for order in @orders %>
  <tr>
    <td align="right"><%= order.id %></td>
    <td align="right"><%= order.status %></td>
    <td align="right"><%= order.total %></td>
    <td align="right"><%= order.books.size %></td>
    <td align="right"><%= order.created_at.strftime("%Y-%m-%d %I:%M") %></td>
    <td align="right"><%= order.updated_at.strftime("%Y-%m-%d %I:%M") %></td>
    <td><%= button_to "View", :action => 'show', :id => order %></td>
  </tr>
<% end %>
</table>
<%= 'View page:' if @order_pages.page_count > 1 %>
<%= pagination_links @order_pages %>
```

At the top of page, we include the navigation. Then we loop through the orders and display them in a table. We also include a button that takes us to the order details page, which we'll implement shortly. At the very bottom of the page, we use the `pagination_links` helper method to generate a menu that is used to navigate between the pages. Each page displays ten orders, and if there are more, a menu like this will be displayed:

```
View page: 1 2 3 4 5 6 7 8 9 10
```

That was easy! We have now implemented the View Orders user story. Before moving on to the next user story, let's do a quick acceptance test, by opening `http://localhost:3000/admin/order`. If you (or your customers) have created some orders, you'll see something similar to Figure 9-13.

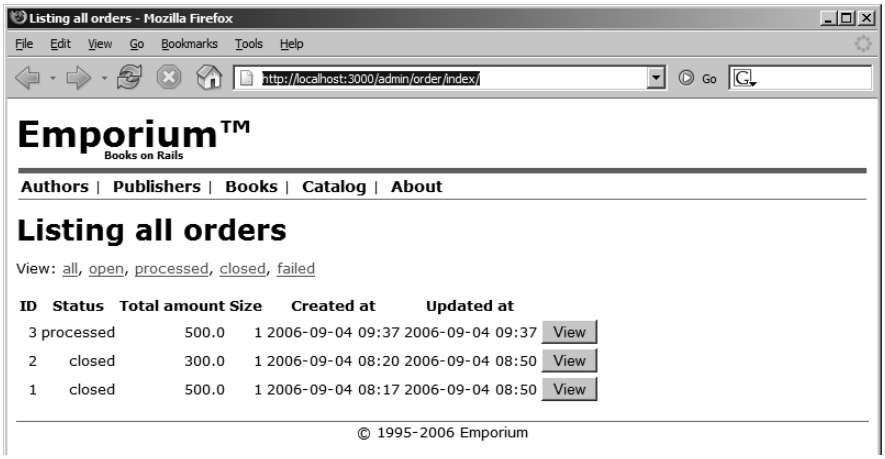


Figure 9-13. *The view orders page displaying all orders*

If you navigate to the page that shows processed orders (`http://localhost:3000/admin/order/processed`), you'll see orders that the system has sent to the payment gateway, but which George still needs to ship to the customer and close. An example of this page is shown in Figure 9-14.

Figure 9-14. *The view orders page showing only processed orders*

■ **Note** For the administrator user stories, we won't walk through functional and integration tests. At this point, you should be familiar with developing these tests and able to write them all by yourself.

Implementing the View Order User Story

The order details page is where George can see the information about the order. He needs to check this page before he ships the books to the customer. The page will show the customer's contact information and shipping address. Furthermore, we need to display the order details—in other words, the books that the customer has ordered. The credit card details are needed only by the payment gateway, and keeping them in our database would be a huge security risk.

Open `app/controllers/order_controller.rb` and replace the empty `show` action with the following code:

```
  def show
    @page_title = "Displaying order ##{params[:id]}"
    @order = Order.find(params[:id])
  end
```

The action simply pulls out the specified order from the database and sets the page title. Next, create the view by saving the following code in `app/views/admin/order/show.rhtml`:

```
<%= render :partial => 'navigation' %>
<p>
  Order total $<%= @order.total %>
</p>
<h2>Contact Information</h2>
<dl>
  <dt>ID</dt>
  <dd><%= @order.id %></dd>
  <dt>Email</dt>
  <dd><%= @order.email %></dd>
  <dt>Phone number</dt>
  <dd><%= @order.phone_number %></dd>
</dl>
<h2>Shipping Address</h2>
<dl>
  <dt>First name</dt>
  <dd><%= @order.ship_to_first_name %></dd>
  <dt>Last name</dt>
  <dd><%= @order.ship_to_last_name %></dd>
  <dt>Address</dt>
  <dd><%= @order.ship_to_address %></dd>
  <dt>City</dt>
  <dd><%= @order.ship_to_city %></dd>
  <dt>Postal/Zip code</dt>
  <dd><%= @order.ship_to_postal_code %></dd>
  <dt>Country</dt>
  <dd><%= @order.ship_to_country %></dd>
</dl>
```

```
<h2>Order Details</h2>
<% for item in @order.order_items %>
  <%= link_to item.book.title, :action => "show",
        :controller => "catalog", :id => item.book.id %>
  <%= pluralize(item.amount, "pc", "pcs") %>,
  $<%= item.price * item.amount %></br>
<% end %>
<p>
  <%= button_to "Close Order", :action => 'close', :id => ➥
@order unless @order.closed? %>
</p>
```

At the top of the page, we show the navigation and order total. Next, we show the contact information, shipping address, and order details sections. At the bottom, we display a button that allows George to close the order, but only if the order hasn't been closed already, which is checked by calling the `closed?` method on the `Order` model. Note that the page will generate an error until we have added the code shown here to the `Order` model (`app/models/order.rb`):

```
def closed?
  status == 'closed'
end
```

Let's perform an acceptance test. Click the View button from the view orders page (shown in Figures 9-13 and 9-14). You should now see the order details, as shown in Figure 9-15.

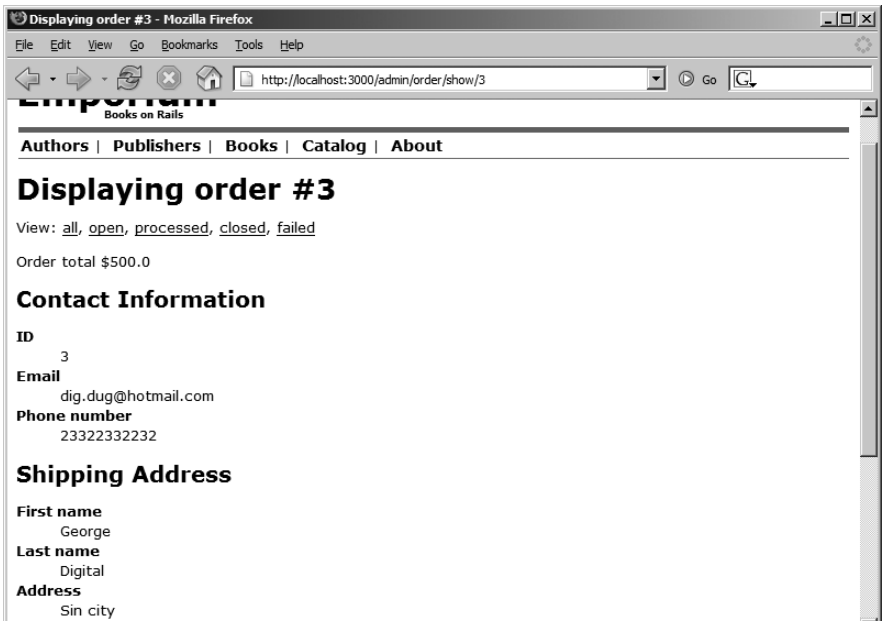


Figure 9-15. *The order details page displaying a test order*

Implementing the Close Order User Story

The Close Order user story is the last one we'll implement in this sprint. It is used by George (after he has shipped the order) to set the order status to `closed`. The user story requires that we change the `close` action in `order_controller.rb` as follows:

```
def close
  order = Order.find(params[:id])
  order.close
  flash[:notice] = "Order #{order.id} has been closed"
  redirect_to :action => 'index', :id => 'closed'
end
```

The code finds the specified order and calls the `close` method on the `Order` model (`app/models/order.rb`):

```
def close
  self.status = 'closed'
  save!
end
```

This method sets the status to `closed` and saves the order. After this, the action sets a flash message and redirects to the Closed section of the view orders page.

Test the Close Order user story by closing an order. View the details of a processed order (click the View button for the transaction on the view orders page), and you should see the Close Order button at the bottom of the order details page, as shown in Figure 9-16.

Click the Close Order button, and you are redirected to the page shown in Figure 9-17.

This ends our implementation of the checkout and order-processing functionality. However, you still need to take into account two other items when processing orders: shipping costs and taxes. We'll take a brief look at those calculations next.

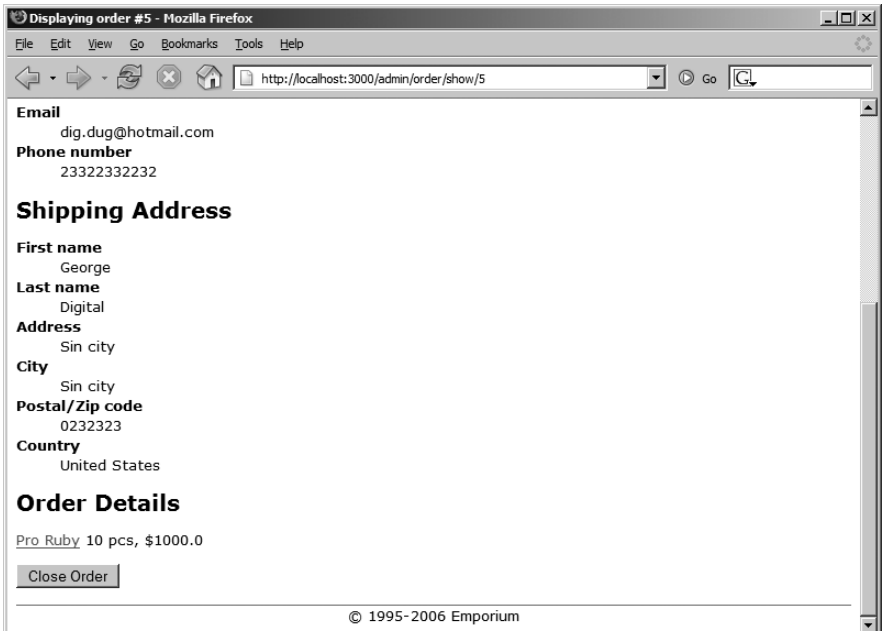


Figure 9-16. *The order details page displaying the Close Order button*

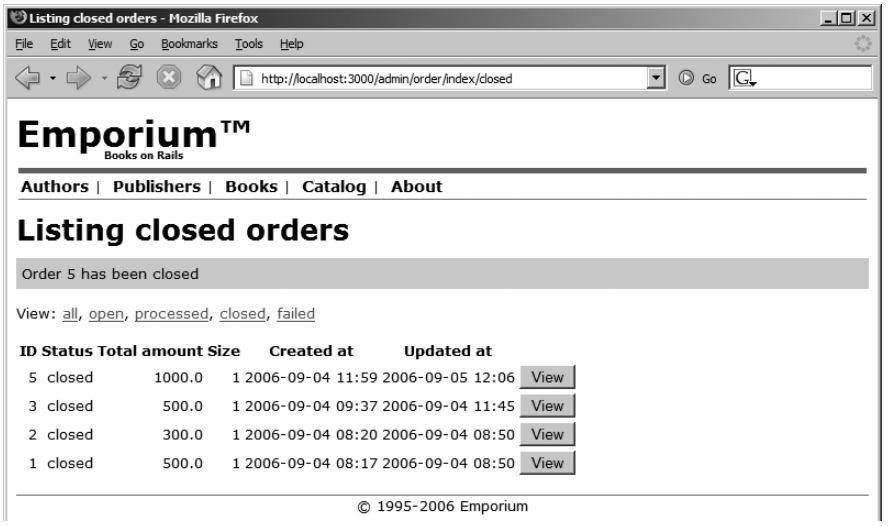


Figure 9-17. *The view orders page displaying a message after closing an order*

Calculating Shipping Costs and Taxes

For calculating Federal Express (FedEx) and United Parcel Service (UPS) shipping costs, you can use the handy Shipping RubyGem (`http://shipping.rubyforge.org/`).

Using the Shipping Gem

You'll first need to register an account at FedEx (`http://www.fedex.com`) and UPS (`http://www.ups.com`).

Next, install the Shipping gem by executing this command:

```
$ sudo gem install shipping --include-dependencies
```

Then save your account information in `config/shipping.yml`:

```
fedex_url: https://gatewaybeta.fedex.com/GatewayDC
fedex_account: <your FedEx account number>
fedex_meter: <your FedEx meter number>

ups_account: <your UPS account number>
ups_user: <your UPS username>
ups_password: <your UPS password>
```

■ **Note** You can obtain the `fedex_meter` number by first registering for a FedEx account, and then calling the `Shipping::FedEx.register` method with the following parameters: `name`, `company`, `phone`, `email`, `address`, `city`, `state`, `zip`, `fedex_account`, and `fedex_url`. You need to register to both the live and test servers. The URL to the live environment is `https://gateway.fedex.com/GatewayDC`.

Now let's test how much it costs to send an order weighing 2.0 pounds from Emporium (located in New York) to Durham, North Carolina. We chose UPS because it's easier to use than FedEx, which requires that we apply for a meter number. Save the code shown here in `test/unit/test_shipping.rb`:

```
require File.dirname(__FILE__) + '/../test_helper'
class PaymentTest < Test::Unit::TestCase
  def test_ups_shipping
    params = {
      :zip => 27712,
      :state => "North Carolina",
      :sender_zip => 10001,
      :sender_state => "New York",
      :weight => 2,
      :prefs => '../../config/shipping.yml'
    }
    ship = Shipping::UPS.new params
    assert ship.price > 5
    puts ship.price
  end
end
```

Run the test by executing the following command:

```
$ ruby test/unit/test_shipping_rb
```

The output when we ran it was $8.59 (your result might be different).

Calculating Taxes

Taxes vary from country to country and from state to state. Check your local tax system for how to calculate taxes.

As an example, if you live in the US and buy goods from an out-of-state store, then you are not required to pay sales tax. But, if you buy goods from a store located in your own state, the store charges you a sales tax, which is different for each state. Some goods are also exempt from sales taxes. See `http://en.wikipedia.org/wiki/Sales_taxes_in_the_United_States` for more information about sales taxes in the US. For general information about sales taxes, see `http://en.wikipedia.org/wiki/Sales_tax`.

In Europe, most countries use a tax system called value-added tax (VAT). The VAT rate is different in each country. For example, in Finland, the standard VAT rate is 22%. See `http://en.wikipedia.org/wiki/Value_added_tax` for more information about VAT.

Summary

In the first part of this chapter, we built a checkout page that can be used by Emporium's customers for placing orders. We also demonstrated how to bill the customer by integrating with PayPal and Authorize.Net, using the Active Merchant plugin and the Payment gem.

In the second part of the chapter we built an order-processing administration interface, which can be used to view and change the status of orders. At the end, we briefly explained how you can calculate shipping costs and taxes.

In the next chapter, we'll show you how to add support for multiple languages, which is useful for increasing your customer base.

CHAPTER 10

Multiple Language Support

In this chapter, you will learn how to translate your application into multiple languages. Supporting more than one language is essential to the success of an online business. For example, tens of millions of Spanish-speaking Internet users are all potential customers, and making your website accessible to these people will most likely increase your sales. Supporting a new language usually means a lot of extra work, but we'll show you how to do it easily with the help of a Rails plugin called Globalize.

Getting the Localization Requirements

George is hoping that making the new Emporium website accessible in multiple languages will boost his sales and profits into the stratosphere. His end goal is to make Emporium the biggest online bookstore, and translating the site to Swedish is the first step. He also has plans on expanding to China and North Korea, but that's going to be done later after he has conquered the English- and Swedish-speaking countries.

We tell him that the technical part is easy with Rails, because there's a plugin called Globalize that does the tricks needed for supporting different languages. All he has to do is find someone who can type in the translations, and we'll provide the technical implementation.

For this sprint, we define one user story related to the user interface and four user stories related to the administration interface for managing translations:

- *Change locale*: The customer must be able to change the locale on the Emporium website. This should be done by clicking a link, which changes the locale and stores the setting in the session.
- *List translations*: The administrator must be able to view a list of all text that is used in Emporium. The administrator should be able to easily see text that hasn't been translated and text that has been translated.
- *Add translation*: The administrator must be able to add a new translation for text.
- *Edit translation*: The administrator must be able to edit translated text.
- *Delete translation*: The administrator must be able to delete translated text.

We'll get started by installing the Globalize plugin and seeing how it supports localization.

INTERNATIONALIZATION AND LOCALIZATION

Adapting an application to a specific language or region is commonly referred to as internationalization (i18n) and localization (L10N). The numbers, 18 and 10, specify the number of characters that have been left out in the somewhat cryptic abbreviations.

Internationalization refers to the process of modifying an application's design so that it can support locale differences like text orientation, currency, date and time format, sorting, and so forth. This can be done by externalizing text strings into files or a database, and by developing currency and date formatting utilities.

Localization means adapting an application to a specific language or locale; for example, by translating text into multiple languages. A locale is identified by the user's language and country, and specifies how, for example, numbers, currencies, and dates are displayed on the screen. The code for the US English locale is en-US. Locales are specified by RFC 3066 and consist of two parts. The first is an ISO 639 language code and uses lowercase letters. The second is usually an ISO 3166 country code in uppercase letters.

Using the Globalize Plugin

Because of its close integration with Rails, the Globalize plugin provides a simple way of localizing the following in your application:

- Dates, times, numbers, and currencies. For example, 100000000.00 localized to US standards is 100,000,000.0.
- Text stored in a database. For example, the book *The Old Man and the Sea* is *Der alte Mann und das Meer* in German.
- Text used in views. For example, the link Next Page is Nästa Sida in Swedish.

But that's not all. Currently, Globalize comes with built-in data for 7599 languages and 239 countries, including pluralization rules and language names. Globalize knows, for example, that Tuesday is tisdag in Swedish. Pluralization is done by specifying both the singular and plural form for a translation; for example, "Displaying one book" and "Displaying many books."

■ **Note** Globalize is not the only alternative for localizing Ruby on Rails applications. There's also the GetText library, which allows you to externalize text into resource bundles and translate them into multiple languages. GetText doesn't have as many features as Globalize and lacks support for things like date and currency formatting. For more information about GetText and other options, see `http://wiki.rubyonrails.org/rails/pages/InternationalizationComparison`.

To install Globalize, execute the `script/plugin` script:

```
$ script/plugin install http://svn.globalize-rails.org/svn/globalize/globalize/trunk
```

The script checks out the latest version of the plugin from the Subversion repository and installs it in the `vendor/plugins` directory. By default, the directory is named `trunk`. You should give the directory a more descriptive name. For example, we renamed it to `globalize-for-1.1`.

■ **Note** At the time of writing, the latest version of Globalize didn't support Rails 1.1. Instead of installing the latest version, we had to install the Rails 1.1 specific branch of Globalize with the following command: `script/plugin install http://svn.globalize-rails.org/svn/globalize/globalize/branches/for-1.1`.

Globalize extends the Rails framework's ActionView, ActiveRecord, and ActionMailer modules, allowing it to provide the tools used for translating text. Globalize uses three database tables containing predefined data for thousands of languages and hundreds of countries:

- `globalize_countries`: Holds countries and the country-specific localization rules, including ISO 3166 country code, date formats, and currency formats.
- `globalize_languages`: Holds languages and the language-specific localization rules, including ISO 639 language code, English name, and native name.
- `globalize_translations`: Holds translated text. The table is prepopulated with data for hundreds of languages. The `tr_key` column contains text in the base language. The `text` column is where the actual translation is stored.

Figure 10-1 shows the schema for these tables.

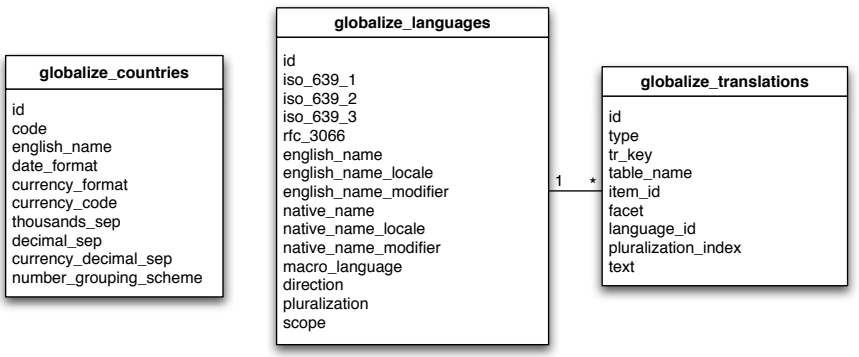


Figure 10-1. *Globalize database schema*

Localizing with Globalize

Globalize uses the term *base language* for the language from which you are translating. So, if you write the text in your views and database in English, this means your base language is English. Globalize uses the base language as the key for other translations. For example, a view containing the following text inserts a record into the `globalize_translations` table, where the text `Picture` is stored in the `tr_key` column:

```
<%= 'Picture'.t %>
```

When translating into Swedish, for example, a new row is inserted, where the `tr_key` contains the text `Picture` and the `text` column contains the translated text `Bild`.

Localizing Text

Using Globalize, you can translate the base language to other languages, by telling Globalize that the text should be translatable. You tell Globalize that text stored in a view should be translatable by appending `.t` (`translate`) to the string, as follows:

```
Locale.set('sv')
puts 'Tuesday'.t
```

```
Tisdag
```

```
Locale.set('fi')
puts 'Wednesday'.t
```

```
Keskiviikko
```

In this example, the base language is English. The first line, `Locale.set('sv')`, sets the locale to Swedish, and the second line prints out Tuesday in Swedish (Tisdag). The fourth line sets the locale to Finnish, and the fifth line prints out Wednesday in Finnish (keskiviikko). This example doesn't require you to translate the weekdays to Swedish and Finnish, because Globalize already contains these translations. However, if you want to add translations, use `Locale.set_translation` and `Locale.set_pluralized_translation`, as explained in the Globalize API documentation (`http://globalize.rubyforge.org/`).

All the work is done behind the scenes by Globalize. The translated text is pulled out of the database with a query similar to this one:

```
SELECT * FROM globalize_translations WHERE ➥
(tr_key = 'Tuesday' AND language_id = 6024 AND pluralization_index = 1) ➥
AND ( (globalize_translations.`type` = 'ViewTranslation' ) ) LIMIT 1
```

Globalize caches the view translations in memory to avoid unnecessary database hits.

Localizing the Model

You can translate text stored in your database by adding a call to the `translates` method to your model, followed by the column names that should be translatable, as follows:

```
class Book < ActiveRecord::Base
  translates :title, :blurb
end
```

This tells Globalize that you want to be able to translate the `title` and `blurb` columns in the `books` table. After you have translated the text and stored the translations in your database, you can execute, for example, the following code with the `script/console` command to print out the text in different languages:

```
book = Book.find_by_name('Bodo Bear on the Farm')
Locale.set('de-DE')
book.reload
puts book.title
```

```
Bodo Bär auf dem Bauernhof
```

In this example, the base language is English. The first line finds the record using the base language, and then sets the locale to German and prints out the book title in German. Notice that you don't need to add the `.translates` method call.

As with view translations, Globalize works behind the scenes and modifies the SQL query on the fly by intercepting calls to the `find` methods (except for the `find_by_sql` method). The modified SQL query uses a left outer join to pull in the translations from the `globalize_translations` table, as follows (this is just part of the query):

```
...LEFT OUTER JOIN globalize_translations AS...
```

This can potentially have a serious impact on the performance, so watch the MySQL slow query log for problems. Note that Globalize doesn't cache the model translations, in contrast to the view translations, which are cached.

■ **Note** The MySQL slow query log is your friend when you're having problems with slow queries. Enable it and watch the log for queries that can be optimized. For more information about the MySQL slow query log, see `http://dev.mysql.com/doc/refman/5.0/en/slow-query-log.html`.

Globalize also has a piggybacking feature that allows you to retrieve translations for associated objects in the same query. To enable this feature, use the `include_translated` option:

```
Book.find(:all, :include_translated => Product)
```

Localizing Dates, Currencies, and Numbers

Globalize provides localization features for dates and numbers. These classes can be localized by using the `localize` method:

```
puts Time.now.localize("%d %B %Y")
```

```
05 July 2005
```

```
12345.45.localize
```

```
"12,345.45"
```

The first line in the sample output prints out the current date, which is localized according to the rules for the base language by calling the `localize` method. The third line demonstrates how to localize a number in the same way. As you can see, Globalize converts the number `12345.45` to the localized string `"12,345.45"`.

Currencies are formatted automatically according to the current locale by adding the following to the model:

```
class Item < ActiveRecord::Base
  composed_of :price_localized, :class_name => "Globalize::Currency",
      :mapping => [ %w(price cents) ]
end
```

By calling `composed_of`, you are telling ActiveRecord that the new `price_localized` field is of type `Globalize::Currency`, and that the value for it is taken from the `price` field, which is specified in cents, since the `Globalize::Currency` constructor expects this. Creating a book where the `price` field has the value 100000 and the locale was set to en-US prints out the string `$1,000.00`.

■ **Note** The `price_localized` field is not editable because of a limitation in Globalize that might be fixed in the future. This is why we can't use it in a form and need to create a separate field for it.

UPDATING THE BASE LANGUAGE

Be careful when updating the base language, which is used in the views and database, because this breaks the link between the base language and the translations. For instance, suppose that you have a view containing the base language text "Next page" and that you have translated it to Swedish. Now, if you change the base language text to "Show next page," you will need to translate the text again to Swedish, and you will still have the old translation in the database. This is a problem in most applications where you want to be able to update the base language text and still keep the translations.

One way of avoiding this problem is not to change the base language after you have translated it to other languages. A better option is to use a base language that is *not* one of languages that you want to support (Swahili maybe?), and then never show it to your users by setting the default locale (in the `config/environment.rb` file) to one of the languages that you want to support. Just use the base language as the key for your translations. So, instead of writing the full text in the view, just write a summary, such as "help_page_text," and translate the text as usual to the target languages. But remember that the users should never see the base language text, because you've set the default locale to another language.

Setting Up Globalize

As previously mentioned, Globalize uses three database tables. After you've installed Globalize, as described earlier, run the following command to update your database schema and import the data:

```
$ rake globalize:setup
```

The command can take a minute or two to run, as it inserts a lot of data.

Next, you need to update the application's environment configuration to include Globalize. This will initialize Globalize at startup. You should also set the base language and the default locale for your application in the same configuration file. To do all this, add the following code to the end of `config/environment.rb`:

```
# globalize
include Globalize
Locale.set_base_language 'en-US'
DEFAULT_LOCALE = 'en-US'
```

The default locale is a separate setting, since it does not necessarily need to be the same as the base language. We are including Globalize in the default namespace with the `include Globalize` call, so that we can use `Locale.set` instead of `Globalize::Locale.set`.

Implementing the User Stories

With Globalize set up, we can now start localizing the Emporium application. But first, we'll change the base controller so that users can change the locale.

■ **Note** We won't be following TDD very strictly in this chapter, as we want to concentrate on showing you how to use the Globalize plugin.

Implementing the Change Locale User Story

The base language, or default locale, for Emporium is US English. The user should be able to change to another language by clicking a link. For example, by accessing the following URL, the user can change the locale to British English:

```
http://localhost:3000/forum?locale=en-GB
```

Accessing the following URL changes the locale to Swedish:

```
http://localhost:3000/forum?locale=sv
```

Note that the link contains the `locale` parameter, which tells your system which locale the user wants to use. The code that changes the locale is easily implemented as a `before_filter`. As explained in Chapter 5, filters allow you to run code before, around, and after the controller's action is called. In our case, the `set_locale` filter (Listing 10-1) is called before anything else is called.

Next, open `app/controllers/application.rb` in your editor and add the code for the `set_locale` method shown in Listing 10-1.

Listing 10-1. *The set_locale Filter*

```
class ApplicationController < ActionController::Base
  before_filter :set_locale

  private

  def set_locale
    accept_lang = request.env['HTTP_ACCEPT_LANGUAGE']
    accept_lang = accept_lang.blank? ? nil : accept_lang[/[^,;]+/]

    locale = params[:locale] || session[:locale] || accept_lang || DEFAULT_LOCALE
```

```
    begin
      Locale.set locale
      session[:locale] = locale
    rescue
      Locale.set DEFAULT_LOCALE
    end
  end
end
```

The filter looks for a user-specified locale from the request parameters. If the locale is not found in the request parameters, it looks for a locale stored in the session. If the locale is not found in either the request or session, it tries to use the `HTTP_ACCEPT_LANGUAGE` header, which is sent out by the browser. Note that the algorithm falls back on the default locale, which we defined in the `environment.rb` configuration file, if all other methods of discovering the user's locale fail.

■ **Tip** To set the `HTTP_ACCEPT_LANGUAGE` HTTP header in Firefox, select Tools ➤ Options and click the Advanced icon at the top of the Options dialog box. Then click the Edit Languages button on the General tab, as shown in Figure 10-2.

Lastly, the filter stores the locale in the session, so users don't need to change the locale every time they navigate to a new page. The filter falls back on the default locale if the user specifies an invalid locale.

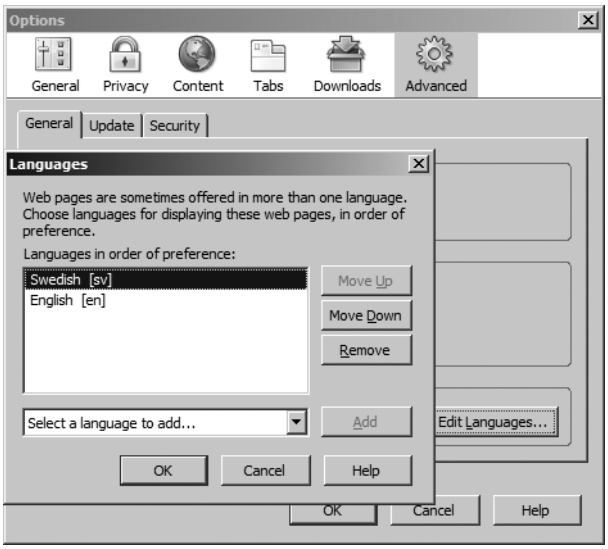


Figure 10-2. *Setting the preferred language with the HTTP_ACCEPT_LANGUAGE header*

Implementing the Translation User Stories

Jill, Emporium's best customer, has been recruited by George to translate and localize Emporium to Swedish, and later to Chinese and other languages. Jill should be able to access a page that lists all text that can be translated. Text that hasn't been translated should be highlighted, so she can see it clearly without using a magnifying glass. The page should allow her to add text and the translations for it directly. She will also need to be able to manage existing translations.

These requirements are covered by the List Translations, Add Translation, Edit Translation, and Delete Translation user stories. These user stories make up the administration interface that can be used to manage the view and model translations. We'll start by implementing the List Translations user story.

Implementing the List Translations User Story

This translation list page should show all translations available in the database. First, use the `generate` script to create the controller and one view:

```
$ script/generate controller 'admin/translate' index
```

```
      exists  app/controllers/admin
      exists  app/helpers/admin
      create  app/views/admin/translate
      exists  test/functional/admin
      create  app/controllers/admin/translate_controller.rb
      create  test/functional/admin/translate_controller_test.rb
      create  app/helpers/admin/translate_helper.rb
      create  app/views/admin/translate/index.rhtml
```

Note that we don't tell the `generate` script to create actions for the other user stories, because these user stories don't need a dedicated view.

Next, change `app/controllers/admin/translate_controller.rb` as follows:

```
class Admin::TranslateController < ApplicationController
  def index
    @page_title = ➥
"Translating from #{Locale.base_language} to #{Locale.language.english_name}"
    @view_translations = ViewTranslation.find(:all,
      :conditions => [ 'language_id = ?', Locale.language.id ], :order => 'id desc')
  end
end
```

The `index` action uses the `ViewTranslation` model, which is provided by Globalize, to retrieve all translations from the database. The list is filtered so that only the current locale is shown. The list is also sorted, so that the latest inserted translations are shown first. We set the page title to show the base language and the language to which we are translating, such as "Translating from English to Swedish."

■ **Note** The code described here is originally from the Globalize wiki, but has been modified and improved.

Next, create the `index` action's view, which displays the translations in a table and uses an Ajax-enabled in-place editor to allow the user to edit the translation, by saving the code shown in Listing 10-2 in `app/views/admin/translate/index.rhtml`.

Listing 10-2. *The Translation View*

```
<%= form_tag :action => 'create' %>
<p><label for="text">Text</label><br />
<%= text_field 'view', 'text' %></p>
<p><label for="translation">Singular form</label><br />
<%= text_field 'view', 'singular_form' %>
<p><label for="translation">Plural form (optional)</label><br />
<%= text_field 'view', 'plural_form' %></p>
<%= submit_tag "Add translation" %>
<%= end_form_tag %>

<table style="width: 100%;">
  <tr>
    <th>id</th>
    <th>key</th>
    <th>qty</th>
    <th>translation</th>
  </tr>
  <% @view_translations.each do |tr| %>
  <tr id="row_<%= tr.id %>">
    <td><%= tr.id %></td>
    <td><%= tr.tr_key %></td>
    <td><%= tr.pluralization_index %></td>
    <td>
    <span id="tr_<%= tr.id %>" <%= 'class="translate"' if tr.text.nil? %>>
      <%= tr.text || 'Click here to translate' %>
    </span>
    </td>
    <td>
   <%= link_to_remote "Delete",
      :url => { :action => "destroy", :id => tr },
      :confirm => "Are you sure you want to delete '#{tr.id}'?",
      :post => true
    %>
    </td>
```

```
  </tr>
  <%= in_place_editor "tr_#{tr.id}",
        :url => { :action => :set_translation_text, :id => tr },
        :load_text_url => ➥
url_for({ :action => :get_translation_text, :id => tr }) %>
  <% end %>
</table>
```

The first part of the view contains a form for the Add Translation user story. This form can be used to add new translations directly to the database. The form has three fields: the text that should be translated, the singular form of the translated text, and the plural form of the translated text. The plural form is optional.

For example, suppose you entered the following in the form:

- Text: `Displaying %d books`
- Singular form: `Displaying one book`
- Plural form: `Displaying %d books`

Then you could use this code to print out the singular and plural form of the text using the `irb` command:

```
"Displaying %d books" / 1

"Displaying one book"

"Displaying %d books" / 2

"Displaying 2 books"
```

Next, the list displays the internal `id`, the base language text (`tr_key`), the pluralization index (`1` for singular and `0` for plural form), and the translated text. Notice that there's also a Delete link that can be used to delete translations from the list.

At the end of the view, there's an in-place-editor that is created with the `in_place_editor` helper. This helper creates a form, which uses Ajax to update the edited field's contents without refreshing the whole page.

To make it easier for Jill, the administrator, to notice text that hasn't been translated yet, you should add the following new CSS class to `public/stylesheets/style.css`:

```
.translate {
  font-weight: bolder;
  color: red;
}
```

This will highlight text that hasn't been translated in bold and red.

Now that you have set up the translation view, open `http://localhost:3000/admin/translate?locale=sv` in your browser. You should see a page similar to the one shown in Figure 10-3.

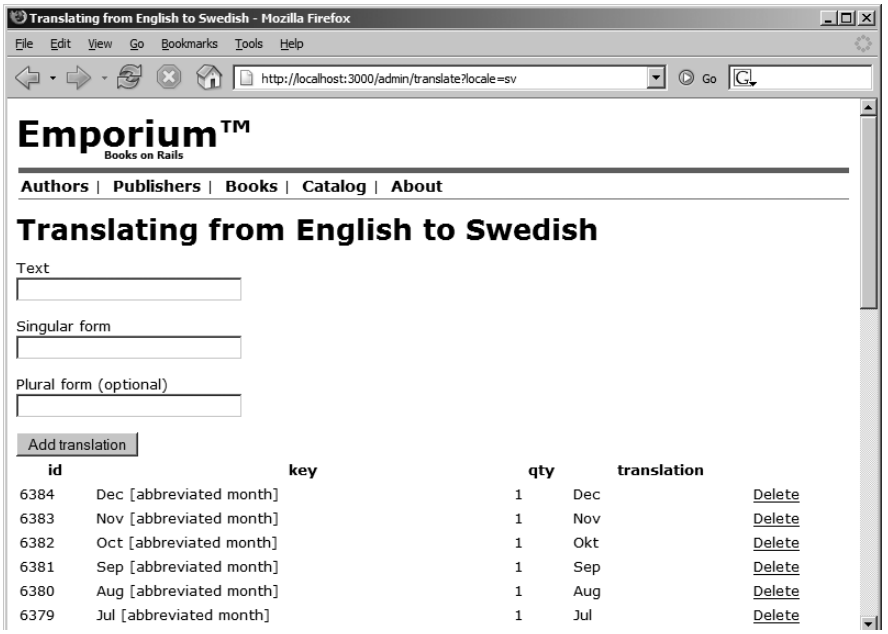


Figure 10-3. *The translation list page*

The list already displays the default translations that Globalize inserted when you ran `rake globalize:setup`.

If you click the Add translation button, you should get an "Unknown action" error. This is because we haven't implemented the Add Translation user story yet. We'll do that next.

Implementing the Add Translation User Story

Add the following code (showing the `create` action) to `translate_controller.rb`:

```
  def create
    from = params[:view][:text]
    singular = params[:view][:singular_form]
    plural = params[:view][:plural_form]
    if(plural.empty? && !singular.empty? && !from.empty?)
      Locale.set_pluralized_translation(from, 1, singular)
      flash[:notice] = "Translated '#{from}' to '#{singular}'"
    elsif(!plural.empty? && !singular.empty? && !from.empty?)
      Locale.set_translation(from, Locale.language, singular, plural)
      flash[:notice] = ➥
"Translated '#{from}' to singular '#{singular}' and plural '#{plural}'"
    else
      flash[:notice] = ➥
"Please specify singular and/or plural form for the translation"
    end
    redirect_to :controller => 'translate', :action => 'index'
  end
```

The `if` clause checks if the user is entering a translation that has only a singular form, and then uses `Locale.set_pluralized_translation` to add the translation to the database. The first parameter is the base language text, the second parameter specifies that the text is in singular form (`1` for singular and `0` for plural), and the last parameter specifies the translated text in singular form.

The `elsif` clause uses `Locale.set_translation` to add a new translation to the database, which has both singular and plural forms. The method's first argument is the base language text, the second is the current locale, the third is the singular form, and the fourth is the plural form of the translated text.

The controller shows an error message if no text was entered in both the `singular` and `text` fields. At the end of the action, we redirect the user to the same translation list page.

Type "Next page" in the Text field and "Nästa sida" in the Singular form field, and then click Add translation. This time, it should succeed, and you should see the list showing the new translation at the top of the list, as shown in Figure 10-4.

You could now try to click the "Nästa sida" text, but the in-place editor would show an error message, because we haven't created the action for it yet.

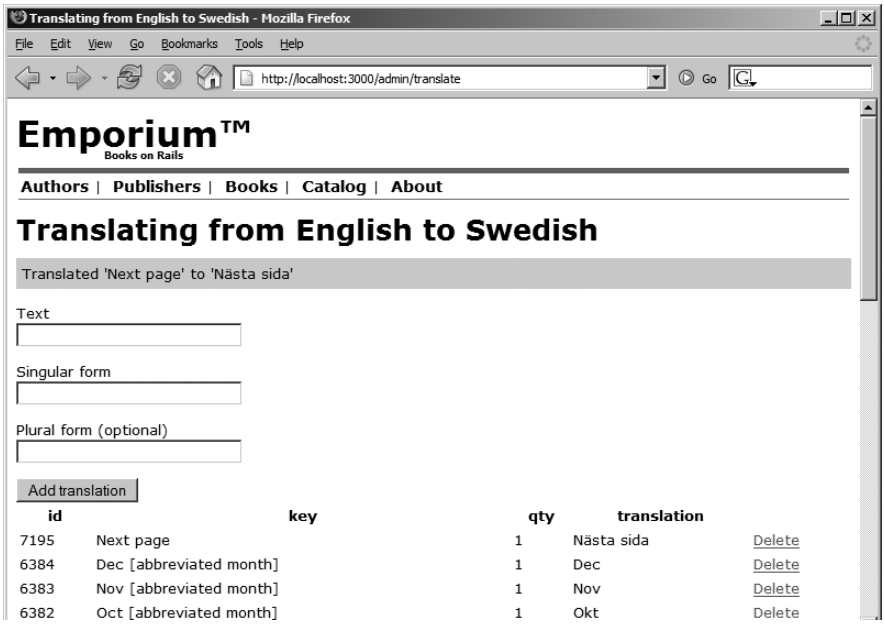


Figure 10-4. *Translation view showing a list of translations*

Implementing the Edit Translation User Story

The in-place editor uses two actions that are called through Ajax requests: `get_translation_text` and `set_translation_text`. Next, add these two actions to the controller (`app/controllers/admin/translate_controller.rb`):

```
def get_translation_text
  @translation = ViewTranslation.find(params[:id])
  render :text => @translation.text || ""
end

def set_translation_text
  @translation = ViewTranslation.find(params[:id])
  previous = @translation.text
  @translation.text = params[:value]
  @translation.text = previous unless @translation.save
  render :text => @translation.text || '[no translation]'
end
```

The get action is called to retrieve and show the translation for the current record, when the administrator clicks somewhere inside the in-place editor field (in the translation column). The set action is called when the administrator saves the translation by clicking the OK button.

Now clicking the translated text in the translation column opens an in-place editor, which allows you to edit the translation, as shown in Figure 10-5. You can save the changes by clicking the OK button, or cancel them by clicking the Cancel button. You can even open more than one in-place editor at a time.

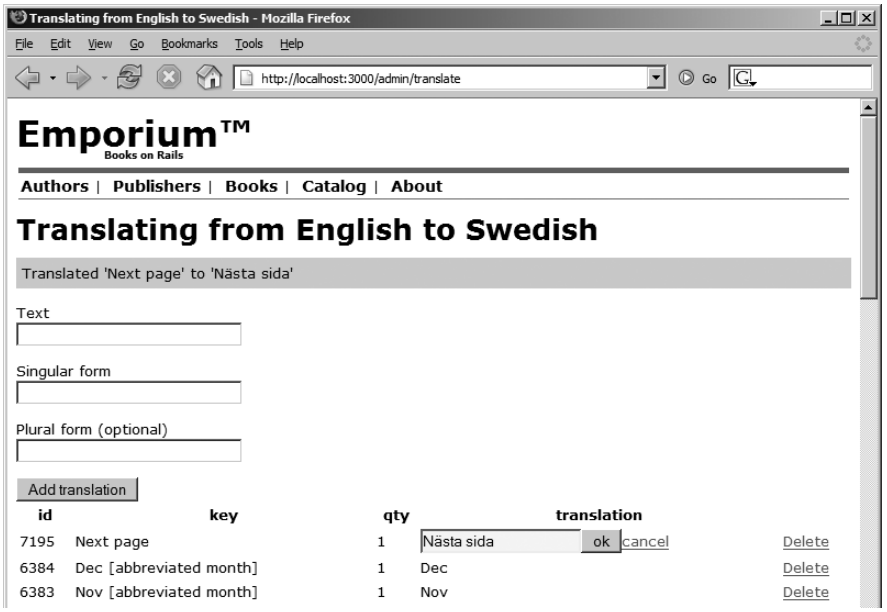


Figure 10-5. *The in-place editor*

Implementing the Delete Translation User Story

We have one user story left to implement: the Delete Translation user story, which is triggered by clicking the Delete link (see Figure 10-5).

Add the following code for the Delete Translation user story to the controller (`app/controllers/admin/translate_controller.rb`):

```
def destroy
  ViewTranslation.find(params[:id]).destroy
  render :update do |page|
    page.remove("row_#{params[:id]}")
  end
end
```

This uses the `ViewTranslation` model to find and delete the specified translation from the database. Like the two actions used by the in-place editor, this action is called asynchronously. After the action has deleted the translation, it uses an `.rjs` script to remove the `tr` element having the specified `id` from the calling page. This deletes, for example,

the `<tr id="row_1">…</tr>` HTML element, including the contents, from the view. Recall that the `.rjs` template, introduced in Chapter 5, generates a JavaScript snippet that is executed within the browser.

Now that everything is in place, you can start translating your views and Book models. We'll first show you how to translate the view.

Translating the View and the Book Model

In this section, we'll show you how to translate the book catalog we created in Chapter 4. We'll translate the view and then the Book model.

Translating the View

To prepare a view for translation, you need to change the text to a Ruby string, surround it with `<%= %>`, and append a call to `.translate` (or its alias `.t`) to the string. For example, to prepare the text "Type a question for help" for translation from the base language to other languages, change it to this:

```
<%= 'Type a question for help'.t %>
```

■ **Note** Globalize inserts a new record into the database (the base language text) for each string that is appended with a `.t` (or `.translate`) method call when that particular code is executed. Code that is located inside an `if` statement that doesn't get executed won't be inserted into the database.

After changing the text, you should switch from the base locale to the locale to which you want to translate. In the Emporium application, this is done by appending `?locale=iso_code` to the URL.

To recap, the required steps for translating a view are as follows:

1. Change the base language text into Ruby strings and add a call to the Globalize `translate` method. For example, `Page title` to `<%= 'Page title'.t %>`.
2. Change to the locale you want to translate the page into by accessing any given page with the correct locale parameter. For example, use `?locale=sv` to change the locale to Swedish.
3. Access the page you want to translate. This inserts a row into the `globalize_translations` table for each translatable Ruby string on the page.
4. Access the translation view (`http://localhost:3000/admin/translate?locale=sv`) and add a translation for the text by changing the `text` column of the rows that were inserted in the previous step.
5. Repeat steps 2 through 4 for each locale and page. You could also write a Ruby script that copies empty translations to all locales. That way, you wouldn't need to repeat the steps for each locale.

If you have a complex page or controller that uses a lot of `if/else` branches, it might be difficult to insert all of the text into the database with the preceding steps. In this case, it is better to manually enter the translations using the translation view we just created. Another option is to write a script that extracts all strings that end with `.t` (`.translate`) and then inserts them into the database. But there are still validation error messages that wouldn't be translated. This is because error messages are generated by the Rails framework, and they are shown only when there's a validation error. To prepare error messages for translation, you could submit an empty form that generates validation errors. This will insert the error messages to the database, so you can translate them.

We are now ready to start modifying the views. First, change the view `app/views/catalog/_books.rhtml` partial as shown here:

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd>
        <strong>
          <%= add_book_link("+", book) %>
        </strong>
      </dd>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small><%= 'Publisher'.t %>: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

Notice that the only thing we changed is the text `Publisher`, which now reads `<%= 'Publisher'.t %>`.

Next, change the pagination links in `app/views/catalog/index.rhtml` as follows:

```
<%= link_to 'Previous page'.t, { :page => @book_pages.current.previous } ➥
if @book_pages.current.previous %>
<%= link_to 'Next page'.t, { :page => @book_pages.current.next } ➥
if @book_pages.current.next %>
```

Notice that we added `.t`, the abbreviation for `translate`, to the previous and next links. This is all you need to do to enable translation for a piece of text in the view.

In the next section, we'll show you how to translate the book title, which is retrieved from the database. This doesn't require any changes in the view; you need to modify only the ActiveRecord model.

Now let's continue by translating the text into Swedish. Access `localhost:3000/catalog?locale=sv` in your browser. (You specify the locale for Swedish by appending `?locale=sv` to the URL.) By accessing the page, we made Globalize insert two new rows: one for "Publisher" and one for "Next page," which we already translated to Swedish. Navigate to the next page by clicking the link. Now you should have one row for "Previous page" also. The three new rows can be found by executing the following query:

```
select * from globalize_translations g order by id desc limit 0,100
```

Note The next and previous links are visible only if there are more than ten rows in the database. This means you need to have at least ten books in the database, because Globalize inserts the rows only when `'Next page'.t` and `'Previous page'.t` are executed. If you haven't already added the fixture files, download them from the Source Code/Downloads section of `www.apress.com`. Once you have copied the fixtures to the `test/fixtures` directory, you can load them into the development database by executing `rake db:fixtures:load FIXTURES=authors,publishers,books,authors_books`.

You can start translating the text by accessing the translation page we created earlier. Open your browser and go to `localhost:3000/admin/translate?locale=sv`. You should see the translation page shown in Figure 10-6. At the top of the translation list, you can see the two rows that are highlighted with the text "Click here to translate."

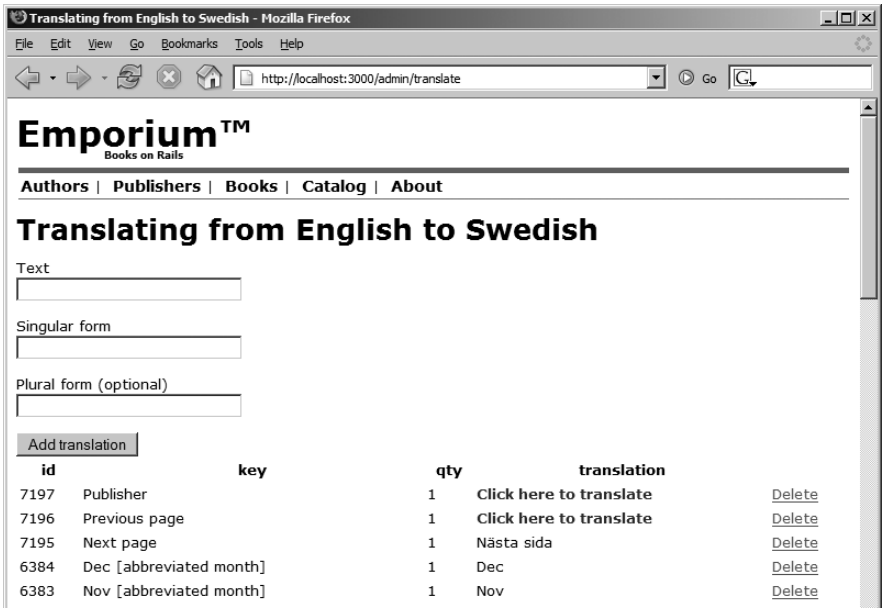


Figure 10-6. *Translation view showing two new rows*

Click the first row's "Click here to translate" text. This opens an in-place-editor, as shown in Figure 10-7. Type in the translation "Förlag," and then click OK. This stores the translation in the database. If you go back to the catalog page, or any other page using the same text, you will see that the text is now displayed in Swedish.

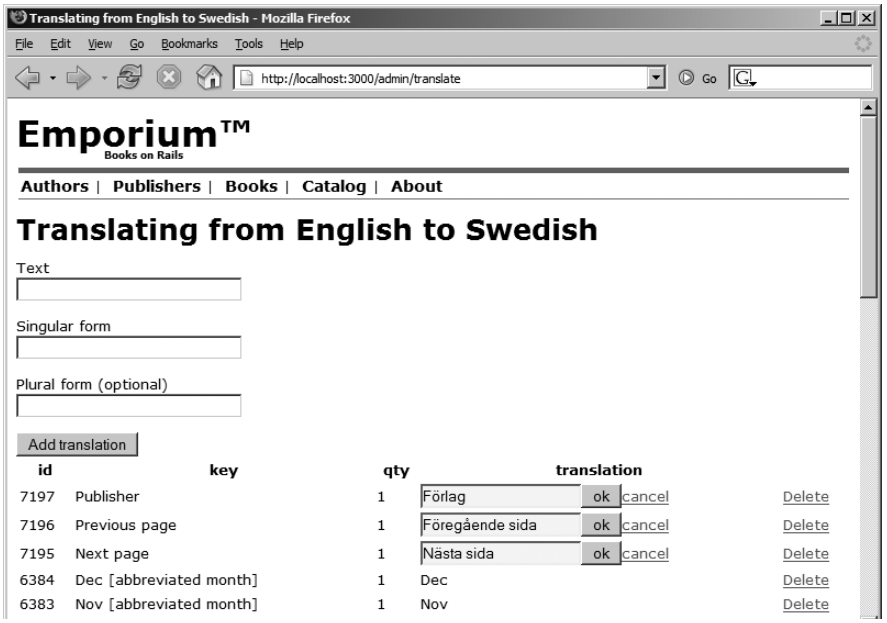


Figure 10-7. *Translation page showing three in-place editors*

■ **Note** If the Scandinavian characters åäö are not displayed correctly, you need to specify a character encoding when starting WEBrick. This is done by specifying the `charset` parameter at startup: `script/server webrick --charset=utf-8`.

Translating the Model

You can prepare an ActiveRecord model for translation by using the `translates` method. Change `app/models/book.rb` as follows:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  translates :title, :blurb
  acts_as_ferret :fields => [:title, :authornames]
```

We added the `:title, :blurb` parameters to the `translates` method. With these, we specify that the `title` and `blurb` database columns should be translated. The translation process for ActiveRecord models works similarly to the view layer, so you need to change `app/views/catalog/_books.rhtml` as highlighted here:

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title.t, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd>
        <strong>
          <%= add_book_link("+", book) %>
        </strong>
      </dd>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small><%= 'Publisher'.t %>: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

■ **Note** You shouldn't need to add a call to the `translates` method in your view, as in our example: `book.title.t`. Adding the `translates` method call to your model should be enough, but at the time of writing, Globalize didn't translate the view, or even add a translation record to the database, until we also changed the view. This is probably a bug and should be fixed.

Now if you access `http://localhost:3000/admin/translate`, you should see the book titles in the list, as shown in Figure 10-8.

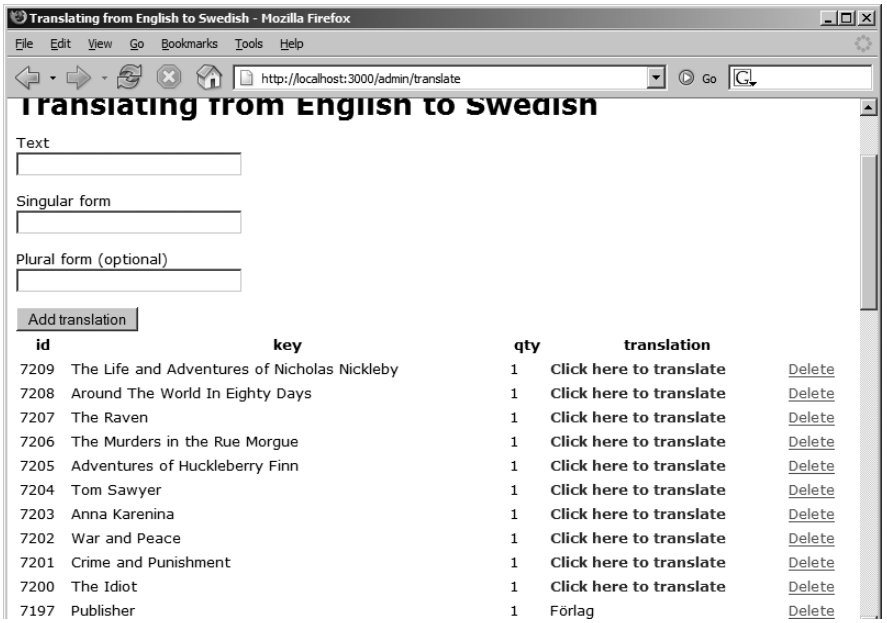


Figure 10-8. *Translation view showing book titles*

Localizing Dates, Numbers, and Currency

As we mentioned earlier, Globalize supports localization of dates, numbers, and currencies. We'll demonstrate how this works on parts of the Emporium site.

Localizing Dates

We'll show you how to localize dates by changing the forum that we created in Chapter 6. Recall that we implemented the View Forum user story, where we display a list of posts, including the date and time when the posts were created. Currently, the format is 01:50:50 2006-07-07, which is not formatted according to the selected locale.

To fix this, open the forum helper `app/helpers/forum_helper.rb` in your editor and change it as follows:

```
module ForumHelper
  def display_as_threads(posts)
    content = ''
    for post in posts
      url = link_to("#{h post.subject}", {:action => 'show', :id => post.id})
      margin_left = post.depth*20
      content << %(
      <div style="margin-left:#{margin_left}px">
        #{url} by #{h post.name} &middot; ➥
#{post.created_at.localize(DEFAULT_DATE_FORMAT)}
      </div>)
    end
    content
  end
end
```

We changed the `display_as_threads` method to localize the date the post was created on.

Add the following line at the end of `config/environment.rb` to complete the localization change:

```
DEFAULT_DATE_FORMAT = '%H:%M:%S - %A %B %d %Y'
```

Restart the server, open your browser, and change the locale to Swedish, by accessing `http://localhost:3000/forum?locale=sv`. The date should now look like this: 01:50:50 - Fredag Juli 07 2006.

The formatting directives for dates and times follow the same rules as `strftime`. Table 10-1 lists some of the more important data and time formatting directives.

Table 10-1. *Some Formatting Directives for Dates and Times*

Code	Description
`%a`	Abbreviated weekday name; for example, Sun
`%A`	Full weekday name; for example, Sunday
`%b`	Abbreviated month name; for example, Jan
`%B`	Full month name; for example, January
`%c`	Preferred local date and time representation
`%d`	Day of the month
`%H`	Hour of the day, using the 24-hour clock format
`%I`	Hour of the day, using the 12-hour clock format
`%j`	Day of the year
`%m`	Month of the year
`%M`	Minute of the hour
`%p`	AM or PM
`%S`	Second of the minute
`%U`	Week number of the current year (count starting from Sunday)
`%W`	Week number of the current year (count starting from Monday)
`%w`	Day of the week
`%y`	Year without century
`%Y`	Year with century
`%Z`	Time zone name

Localizing Numbers and Currencies

Next, we'll show you how to change the product catalog to display the correctly formatted page count and price for the currently selected locale.

Open app/views/catalog/catalog.rhtml in your editor and append .localize to the code that prints the page count:

```
<h2>by <%= @book.authors.map{|a| a.name}.join(", ") %></h2>
<%= image_tag url_for_file_column(:book, :cover_image) ➥
unless @book.cover_image.blank? %>
<dl>
  <dt>Price</dt>
  <dd>$<%= @book.price -%></dd>
  <dt>Page count</dt>
  <dd><%= @book.page_count.localize -%></dd>
```

This will now localize the page count. For example, a book having 400000 pages will be displayed as 400,000.

To localize the price field, we first need to modify the Book model (app/models/book.rb), as shown here:

```
class Book < ActiveRecord::Base
  acts_as_taggable

  has_and_belongs_to_many :authors
  belongs_to :publisher
  has_many :cart_items
  has_many :carts, :through => :cart_items

  translates :title, :blurb
  acts_as_ferret :fields => [:title, :author_names]
  composed_of :price_localized, :class_name => "Globalize::Currency",
      :mapping => [ %w(price cents) ]
```

This adds the new field price_localized to the model, which will be properly formatted when used in a view.

Next, change the catalog view (app/views/catalog/show.rhtml) to use the new field:

```
<h2>by <%= @book.authors.map{|a| a.name}.join(", ") %></h2>
<%= image_tag url_for_file_column(:book, :cover_image) ➥
unless @book.cover_image.blank? %>
<dl>
  <dt>Price</dt>
  <dd>$<%= @book.price_localized -%></dd>
  <dt>Page count</dt>
  <dd><%= @book.page_count.localize -%></dd>
```

Now if you access the book details page (`http://localhost:3000/catalog/show/1`), you should see the following:

```
Price
    $10,000.00
Page count
    400,000
```

Adding Unicode (UTF-8) Support

As we mentioned earlier, George wants to enter the Chinese market sooner or later. In China, the official character set is simplified Chinese, which contains about 6000 different characters. Languages like Simplified Chinese that contain hundreds or thousands of characters do not fit into the standard used by most languages in Europe and North America (ISO-8859-1). Instead, these languages require that we change the application to support UTF-8.

UTF-8 can represent any universal character in the Unicode standard and is backward-compatible with ASCII. UTF-8 is a variable-length character encoding, where each character can be encoded using one to four bytes.

Character encoding is a cause of many problems. For Unicode, or any other character encoding, to work properly, all parts of the system must use the same encoding. In most systems, like Rails-based web applications, this includes the following parts:

- The HTML page
- The HTTP response headers
- The database
- The database connection

Failing to use the correct character encoding in one or more subsystems usually means that Unicode characters are displayed as question marks or some other invalid character. For example, instead of seeing 愶 鬱 執, you would see ??? or some other characters.

■ **Note** The Ruby language doesn't support Unicode out-of-the-box. For example, `String#length` and `String#substr` won't work correctly with Unicode strings. See `wiki.rubyonrails.com/rails/pages/HowToUseUnicodeStrings` for more information about how to add Unicode support to Ruby.

Setting Character Encoding in HTML

Start by changing the HTML page to use the UTF-8 character encoding. Update the layout file, `app/views/layout/application.rhtml`, as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    lang="<%= Locale.active.language.iso_639_1 -%>"
    xml:lang="<%= Locale.active.language.iso_639_1 -%>">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title><%= @page_title || 'Emporium' %></title>
  <%= stylesheet_link_tag "style" %>
  <%= javascript_include_tag :defaults %>
</head>
<body>

<div id="header">
  <h1 id="logo">Emporium&trade;</h1>
```

We specify the character encoding by adding a `Content-Type` `meta` tag to the `head` tag. This will help browsers select the appropriate character encoding while rendering the page.

We also changed the `html` tag so that it declares that the content of the page is written in the language currently selected (`Globalize::Locale.active.language`). This helps search engines in identifying your content and also helps you in positioning your site higher in search results in languages other than English. This is referred to as search engine optimization (SEO).

■ **Tip** Currently, we change the locale by appending a parameter to the URL, such as `/catalog?locale=sv`. This is not optimal from a SEO perspective, as some search engines prefer to index pages without parameters. To fix this, you can add the following to your `config/routes.rb` file: `map.connect ':language/:controller/:action/:id', :defaults => { :language => Locale.language.code, :id => nil }`. Then you can use URLs like `/sv/catalog` or `/de/catalog` instead, which are more likely to be indexed by search engines.

Setting Character Encoding for the HTTP Response

Next, add a new `after_filter` to `app/controllers/application.rb` that sends the correct `Content-Type` HTTP header. Update `app/controllers/application.rb` as follows:

```
class ApplicationController < ActionController::Base
  before_filter :set_locale
  after_filter :set_charset

  private

  def set_charset
    headers["Content-Type"] = "text/html; charset=utf-8" if headers ➥
["Content-Type"].blank?
  end

  def set_locale
    accept_lang = request.env['HTTP_ACCEPT_LANGUAGE']
    accept_lang = accept_lang.empty? ? nil : accept_lang[/[^,;]+/]

    locale = params[:locale] || session[:locale] || accept_lang || DEFAULT_LOCALE
```

Now, when a user accesses a page at the Emporium site, the following response header is sent to the browser:

```
Content-Type: text/html; charset=utf-8
```

This helps the browser in selecting the correct character encoding for displaying your page.

Changing the Database to Use UTF-8

The default character encoding in your database is probably currently latin1 (the default for MySQL). It might be different depending on how you installed MySQL and your system settings. You can check the encoding by opening a `mysql` client and typing in the following command:

```
show variables like 'character\_set\_%';
```

This will show you the character set used by different parts of the system. If you see that the character encoding for the database is not UTF-8, as in the following sample output, you need to fix it by changing the database to UTF-8.

```
$ mysql -uemporium -phacked emporium_development
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 5.0.20-community

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show variables like 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | latin1 |
| character_set_connection | latin1 |
| character_set_database   | latin1 |
| character_set_filesystem | binary |
| character_set_results    | latin1 |
| character_set_server     | latin1 |
| character_set_system     | utf8   |
+--------------------------+--------+
7 rows in set (0.00 sec)
```

If you try to save Unicode text, it will end up being converted to question marks. To use a different character encoding, we need to specify the encoding when we create the database tables. Since we have been using migrations to create the tables, we must modify all of the ActiveRecord migrations. This is done by adding the `options` parameter to the `create_table` method:

```
create_table :publishers, :options => 'default charset=utf8' do |table|
```

MySQL uses `utf8` as the encoding name, instead of the more common format `utf-8`.

After changing the character encoding, re-create the database from scratch with the migrations. And remember to create the globalize tables again with `rake globalize:setup`.

Tip You can also specify the default character encoding used by MySQL at startup by using the `--character-set-server` parameter. After changing the character encoding, all new tables will use the default character encoding, so the migrations don't need to be changed.

The last part of the system that you should change to use the correct encoding is the database connection. Change `config/database.yml` as shown here:

```
development:
  adapter: mysql
  database: emporium_development
  username: emporium
  password: hacked
  encoding: utf8
test:
  adapter: mysql
  database: emporium_test
  username: emporium
  password: hacked
  encoding: utf8
```

Restart WEBrick so that it picks up the changes to the configuration. Remember to add the same setting for the production database connection.

Now access the translation view one last time. Add some Chinese to the Text field, and the translation "Next page" for it, and then click the Add translation button. The Chinese characters should show up correctly in the list.

That's it! George's application is now Globalized, internationalized, and ready for world domination.

Summary

In this chapter, we introduced the Globalize plugin, which can be used to localize an application for multiple locales and languages. We showed you how to install the Globalize plugin and how to create a translation view for easy translation of content. We also showed you how to localize dates, numbers, and text, and how to add Unicode support to your applications.

In the next chapter, we will introduce you to Selenium, Selenium on Rails, and acceptance tests.

CHAPTER 11

Acceptance Testing

In this chapter, you will learn how to automate acceptance tests. Acceptance tests are important, since they prove that the system works according to the requirements.

Throughout this book, George has been performing acceptance testing quite haphazardly by randomly testing that the user stories we have implemented work as he intended. This is better than no acceptance tests at all, but since we're building an e-commerce site, we want to be absolutely sure that everything works all the time. We don't want to lose a customer by providing a service that doesn't work as expected, and George doesn't have the money to hire a dedicated tester. This is where Selenium—an open source testing tool originally developed as an in-house project at ThoughtWorks—comes in handy.

With Selenium and the Selenium on Rails plugin, you can automate acceptance tests that otherwise would be performed manually, or more likely, not at all. You can run the automated acceptance tests when, for example, you refactor code or release a new version. This raises your comfort level by giving you immediate feedback when something breaks.

Using Selenium

The core of Selenium, referred to as Selenium Core, is implemented as JavaScript that runs directly inside your browser, unlike other similar tools like Fitnesse, which run in a separate process. This allows scripts written using Selenium to issue commands like mouse clicks and other actions that mimic the real interaction between a user's browser and a web application. We can, for instance, write a script that simulates George accessing the Emporium forum, filling out the form on the new post page, and then clicking the submit button. Selenium can also check whether the request was successful by, for example, checking that the next page displays the expected content. All these actions are done through Selenium commands, which we'll go through in the "Writing Selenium Tests" section later in this chapter.

Selenium on Rails (`http://www.openqa.org/selenium-on-rails`), a plugin developed by Jonas Bengtsson, integrates Selenium into the Rails framework. The plugin provides many features to simplify the use of Selenium with Rails. For example, it does the following:

- Creates test suites automatically from tests that are located in the same directory. For example, storing a test in `test/selenium/authentication/test_login.sel` would automatically make the test (`test_login.sel`) belong to the `authentication` test suite.
- Deploys Selenium to the test environment automatically. (Selenium is not deployed to the development or production environments by default.)
- Allows you to write Selenium tests in ERB, Selenese, or RSelenese, rather than just HTML.
- Lets you place the Selenium files in a directory other than `/public`. Selenium can be located in `/vendor/selenium` or the RubyGems repository.
- Allows for the use of fixtures in Selenium tests and clearing sessions. Fixtures are run by accessing the URL `/selenium/setup` from acceptance tests.

Before installing the plugin, you need to install Selenium itself, as follows:

```
$ sudo gem install selenium
```

```
Attempting local installation of 'selenium'
Local gem file not found: selenium*.gem
Attempting remote installation of 'selenium'
Updating Gem source index for: http://gems.rubyforge.org
Successfully installed selenium-0.7
```

■ **Note** The RubyGems repository might not contain the latest version. If you want the latest version of Selenium, download Selenium Core from `http://www.openqa.org/selenium-core` and extract it to `vendor/selenium`.

The Selenese format uses the Textile markup format (introduced in Chapter 3), which requires that you have RedCloth installed on your machine. To verify that you have RedCloth installed, execute the following command:

```
$ gem list
```

```
...
RedCloth (3.0.4)
    RedCloth is a module for using Textile and Markdown in Ruby. Textile
    and Markdown are text formats.  A very simple text format. Another
    stab at making readable text that can be converted to HTML.
...
```

If you don't see RedCloth in the list of installed RubyGem packages, execute the following command:

```
$ sudo gem install redcloth
```

Next, install the Selenium plugin directly from the Subversion repository:

```
$ script/plugin install http://svn.openqa.org/svn/selenium-on-rails/
```

This installs the latest version of the plugin in the `vendor/plugins/selenium-on-rails` directory. Execute the `rake`command in the plugin directory to verify that the installation works:

```
$ cd vendor/plugins/selenium_on_rails
$ rake
```

The `rake` script executes all tests. You should see them pass without errors.

■ **Note** On Windows, if you want to use `rake` to run your acceptance tests, you need to install `win32-open3`. See the Selenium on Rails homepage (`http://www.openqa.org/selenium-on-rails/`) for details.

When using Selenium Core, you start the acceptance tests by opening the Selenium test runner in your browser and executing the Selenium test suite. With Selenium on Rails, these steps have been automated and can all be executed with the following command:

```
$ cd /home/projects/george/emporium
$ rake test:acceptance
```

If you execute the command now, Selenium on Rails will complain that you haven't specified the path to the browser executables. To specify the path, you need to modify the Selenium on Rails configuration file (`vendor/plugins/selenium_on_rails/config.yml`). The configuration file allows you to specify where browser executables are located on your system and the environments for which Selenium should be enabled. Create the configuration by renaming

vendor/plugins/selenium_on_rails/config.yml.example to config.yml. Update the paths to match your system's configuration, as in the example shown here:

```
# Enable Selenium for the following environments
environments:
  - test
#  - development
#  - production

# Paths to browsers
browsers:
  safari: '/Applications/Safari.app/Contents/MacOS/Safari'
  firefox: /usr/bin/firefox
  ie: 'c:\Program Files\Internet Explorer\iexplore.exe'
```

■ **Tip** Selenium on Rails runs the acceptance tests in all browsers that have been listed in the configuration file. This makes it easy to test that your application works on multiple browsers. Selenium supports most browsers and platforms, including Windows, Mac OS X, and Linux. Check the Selenium documentation (http://www.openqa.org/selenium-core) to verify that your setup is supported.

Selenium and Selenium on Rails are now installed, which means you can start writing Selenium tests.

Writing Selenium Tests

We'll take a quick look at the Selenium commands and various formats that you can write tests in. After this we'll write an acceptance test for the View Forum user story.

Selenium Commands

Selenium tests have the following basic structure:

|*command*|*target*|*value*|

The items work as follows:

- `command` tells Selenium what to do. For example, you may want to access a page, click a link, and verify that the title of the page is correct. These commands can be classified into the three groups: actions, assertions, and accessors, as described in the following sections.
- `target` tells Selenium on which element to perform the action; for example, you can specify a link or button. The target of an action can be specified with an element locator, which we'll explain after describing the command types.
- `value` is a parameter to the command that tells Selenium, for example, what it should type in a text field.

■ **Tip** The Selenium homepage (`http://www.openqa.org/selenium-core/`) provides a complete reference to all Selenium commands.

Action Commands

Actions perform user actions that modify the state of the application, including clicking links and submitting forms; even drag and drop is supported. So, actions can be used to mimic almost any action performed in a browser by George or any of Emporium's customers. Table 11-1 lists some of the available action commands.

Table 11-1. *Some Selenium Actions*

Name	Description
`open(url)`	Accesses a page, such as the forum[1]
`click(locator)`	Simulates a user clicking a link, button, check box, or radio button[2]
`type(locator, value)`	Simulates a user typing some text into a text box or any other form field that accepts user input
`select(locator, value)`	Selects a specific item from a drop-down list

[1] *The URL must point to the same domain where the Selenium script has been deployed. For example, you can't test* `www.google.com`*. This is due to the so-called same origin policy enforced by browsers. This limitation can be circumvented by using a proxy.*

[2] *Use* `clickAndWait(locator)` *if the action causes a new page to load (as most do).*

Assertion Commands

Assertions, or checks, verify that the state of the application is as expected after one or more actions have been executed by Selenium. For example, you can test that the title of the page or the value of a variable is correct.

Assertions are split into three groups that behave somewhat differently:

- Assertions (`assert`) abort the test if they fail.
- Verifications (`verify`) log the error and allow the test to continue.
- Wait for (`waitFor`) assertions wait for the specified event to happen. They can be used for testing Ajax functionality. For example, you can wait for the value of an element on the page to be updated by an Ajax request.

Table 11-2 shows a partial list of supported Selenium assertions.

Table 11-2. *Commonly Used Selenium Assertions*

Name	Description
`assertLocation (location)`	Asserts the location of the currently loaded page; for example, that it's `/forum` and not `/catalog`
`assertTitle(title)`	Asserts that the title of the currently loaded page is correct
`assertTextPresent(text)`	Asserts that the specified text is present on the page

Verification and wait for assertions use the same format. Use `verifyLocation(/forum)` to verify that the location is `/forum`, and use `waitForLocation(/forum)` to wait for that page to be loaded.

Accessor Commands

Accessors are used for accessing the content that is displayed in the browser and for storing the content in a variable that can be used later in the script. Accessors follow the same logic as verification and wait for assertions. For example, use `storeLocation(variableName)` to store the title of the page in a variable. Table 11-3 lists some commonly used accessors.

Table 11-3. *Commonly Used Selenium Accessors*

Name	Description
`store(value, variableName)`	Stores the value in the specified variable
`storeValue(locator, variableName)`	Stores the value of an input field in the specified variable
`storeText(locator, variableName)`	Stores the text of the specified element in the specified variable
`storeChecked(locator, variableName)`	Stores `true` in the specified variable if the check box is selected, otherwise stores `false`

Let's say that you want to store the text that you entered in the Name field on the post to forum page (`http://localhost:3000/forum/post`) and use the same text in the Subject field. You could do that with the following commands:

```
|open|/forum/post||
|type|post[name]|George|
|storeValue|post[name]|name|
|type|post[subject]|This is posted by ${name}|
```

The first two commands open the post to forum page and type George in the Name field. The third command stores the field's value in a variable (`name`), which is then used on the last row to type the text "This is posted by George" in the Subject field.

Element Locators

Element locators tell Selenium which HTML element a command should be performed against. For example, you can tell Selenium to locate a link where the `id` attribute equals `xyz`. Some of the element locators that Selenium supports are listed in Table 11-4.

Table 11-4. *Some Selenium Element Locators*

Name	Description
`id=element id`	Finds the element with the specified `id` attribute
`name=element name`	Finds the element with the specified `name` attribute
`identifier=element id`	Finds the element that has a matching `id` attribute; if no match is found, finds the element whose `name` attribute matches the specified `id`
`dom=JavaScript expression`	Finds the element using a JavaScript DOM expression; for example, `document.forms['someForm'].someButton`
`xpath=XPath expression`	Finds the element based on an XPath expression; for example, `//a[@href='http://google.com']`
`link=text pattern`	Finds the link that matches the specified pattern

Selenium Test Formats

Selenium acceptance tests can be written in various ways. The default is HTML format. Selenium on Rails also offers the Selenese and RSelenese formats.

HTML Format

Tests using this format are written as normal HTML documents that contain a table:

```
<table>
  <tr>
    <td>open</td>
    <td>/</td>
    <td></td>
  </tr>
  <tr>
    <td>verifyTitle</td>
    <td>Home</td>
    <td></td>
  </tr>
</table>
```

Selenium parses the table and executes the commands in the order they appear.

There is one downside to using HTML tables to write acceptance tests: they are difficult to maintain and write.

■ **Note** You can write dynamic tests by using RHTML templates containing ERB code. However, like HTML, these tests are hard to maintain and write.

Selenese Format

The Selenese format is the easiest to write and maintain. Acceptance tests written using this format contain a table where columns are separated by the pipe character:

```
|open|/help|
|assertTextPresent|The Forgotten Soldier|
|goBack||
```

The only downside to the Selenese format is that the tests are static. This is why Selenium on Rails also supports the RSelenese format.

RSelenese Format

RSelenese scripts are written in Ruby and allow you to use the full power of the Ruby language, as shown in the following example:

```
setup :fixtures => :all
open '/'
assert_title 'Home'
(1..10).each {|i| open :controller => 'catalog', :action => 'list', :page => i }
```

The biggest benefit of using RSelenese is that it allows you to create dynamic tests.

The First Acceptance Test

Now we'll write a simple acceptance test for the View Forum user story, which we implemented in Chapter 6. In the next section, we'll show you how to speed up the process of creating tests by using the Selenium IDE extension for Mozilla Firefox.

Create the first acceptance test by executing the `generate` script:

```
$ script/generate selenium forum/01_view_forum
```

```
      create  test/selenium/forum
      create  test/selenium/forum/01_view_forum.sel
```

Open `01_view_forum.sel` in your editor and modify it to contain the Selenese commands shown here:

```
|open|/forum/|
|assertTitle|Forum|
|assertTextPresent|Forum|
|assertTextPresent|There are no posts|
```

When Selenium executes the View Forum test case, it will perform the following actions in the browser:

- Open the `/forum` URL.
- Verify that the title of the page is Forum.
- Verify that the text "Forum" can be found somewhere on the page.
- Verify that the page contains the text "There are no posts." Note that this requires that the test database is empty. You can empty the table manually or have the script do it, as we'll show you in the next sections.

Selenium tests are stored in the `test/selenium` directory. It is good practice to group related tests into test suites, which is why we put the forum acceptance tests in a folder called `forum`. Selenium on Rails creates test suites automatically by scanning the `test/selenium` directory for subdirectories and acceptance tests. Acceptance tests are sorted by their filenames, which are run sequentially. In our case, the test suite will be created from the `test/selenium/forum` folder.

■ **Tip** If you need the tests to run in a specific order, prepend their filenames with a number. For example, a test suite containing a login and logout test could have files named `01_login.sel` and `10_logout.sel`, so that the login test runs before the logout test. The same naming scheme can be used with test suite directories.

By default, Selenium scripts are enabled only for the test environment. This means you need to start WEBrick in test mode by executing the following command:

```
$ script/server -e test
```

Before running the test, make sure your test database has been updated to the latest version. This can be done with the `rake` command:

```
$ rake db:test:clone_structure
```

You can now execute the test we just created by issuing the following command:

```
$ rake test:acceptance
```

This will open all the browsers that you specified in the Selenium on Rails configuration file, one by one, and execute the View Forum test case in your browser. After the tests have run, you should see the test result page, as shown in Figure 11-1.

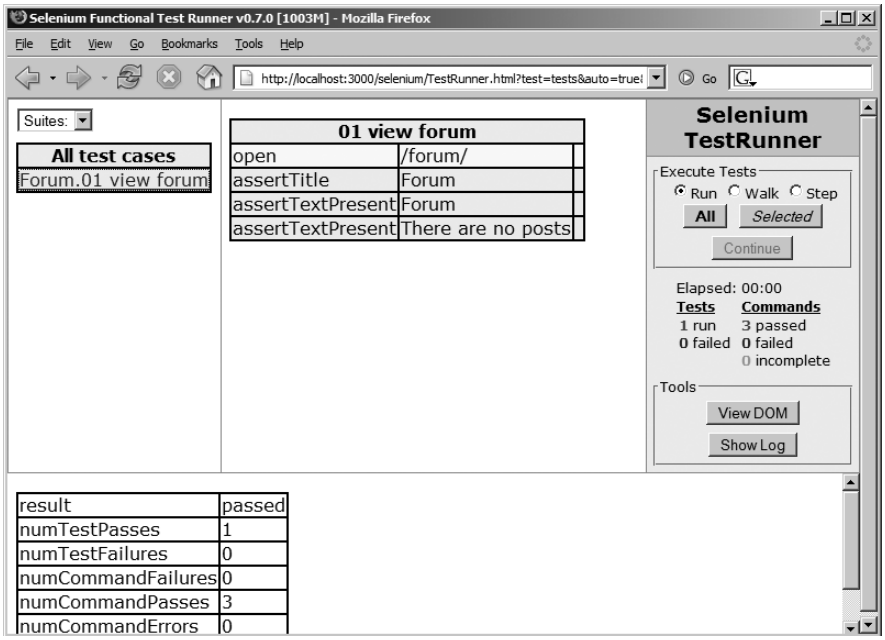


Figure 11-1. *The Selenium TestRunner after a successful test run*

Successfully executed commands are shown in green. Failures are in red. The table at the bottom of the page shows detailed statistics of the test.

Recording Selenium Tests

Writing acceptance tests requires detailed knowledge of how Selenium works (and a lot of typing). The good news is that Selenium IDE simplifies acceptance test creation. Selenium IDE is an integrated development environment that has been implemented as an extension available for Mozilla Firefox only.

Using the Selenium IDE

Selenium IDE allows you to record your tests directly in the browser. You simply click the record button and perform the actions in your browser. Selenium IDE also allows you to run and debug tests. These features are all made possible by the inclusion of Selenium Core in the extension. On top of this, Selenium IDE provides you with features like autocompleting commands and saving recorded tests as HTML, Ruby, or other any other user-defined format.

The Selenium IDE project is hosted, like Selenium itself, at `www.openqa.com`. Install Selenium IDE by going to `http://www.openqa.org/selenium-ide/` and accessing the Selenium IDE Downloads page, as shown in Figure 11-2.

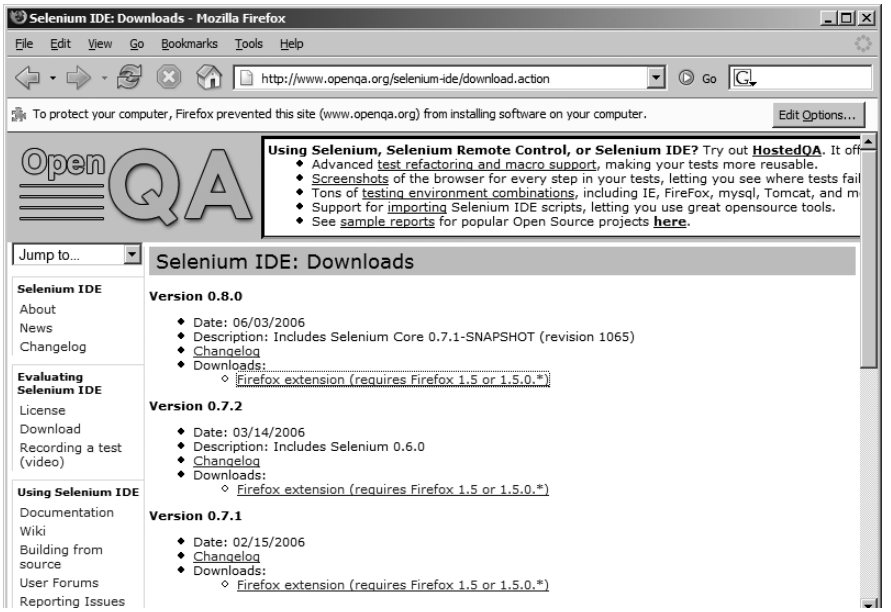


Figure 11-2. *The Selenium IDE Downloads page*

Firefox requires that you allow `www.openqa.org` to install extensions on your machine. So, your first step is to click the Edit Options button at the top left of the Selenium IDE Downloads page (see Figure 11-2) and add `www.openqa.org` to the list of privileged sites.

Next, click the Download link again. This time, you should see the dialog box shown in Figure 11-3. To continue with the installation, click the Install button.

After restarting Firefox, you can open Selenium IDE by selecting Tools ➤ Selenium IDE from the browser menu bar. You should now see the Selenium IDE window, as shown in Figure 11-4.

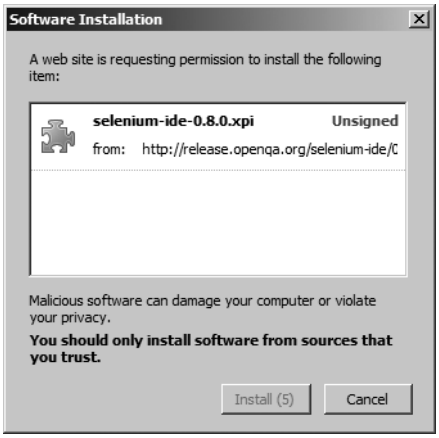


Figure 11-3. *Installing the Selenium IDE extension*

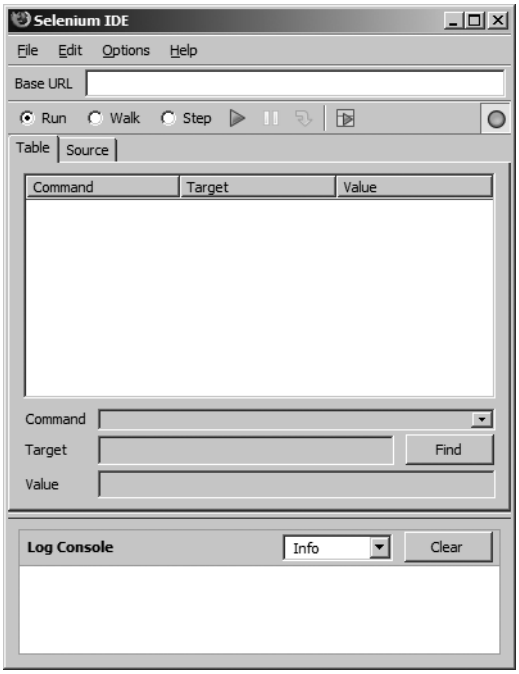


Figure 11-4. *The Selenium IDE window*

The Selenium IDE window has two tabs:

- The Table tab is where the recorded commands are shown. Right-click a row to see a context-sensitive pop-up menu that allows you to insert new commands, delete existing commands, and perform other actions.
- The Source tab is where the actual source for the acceptance test is shown. You can either copy and paste the code or use the File ➤ Save Test menu item to save the test to a file. You can switch between different supported output formats by selecting the appropriate option from the Options ➤ Format menu.

Recording the View Forum Acceptance Test

We'll start by recording a test for the View Forum user story (originally implemented in Chapter 6). This is the same user story that we wrote a simple acceptance test for in the previous section, but this test will check a few different details. Recall that this user story describes how a user is able to view a list of the most recent posts by going to the forum main page, where posts are shown in a threaded fashion:

```
First post
-->Reply 1
---->Reply 1.1
-->Reply 2
```

We will verify that the following requirements are met (in the mentioned order):

- The page title of the forum must be Forum.
- When the forum is empty, the page must show a message and a link that provides instructions on how the user can create a new post.
- When there are more than 20 posts in the forum, the list must be paginated.

Certainly, we could test more aspects of the user story, but this is sufficient for the moment. If something breaks in the future, we can always extend the test.

If not already done, start the Emporium application by executing the following command:

```
$ ruby script/server -e test
```

We have enabled Selenium only for the test environment, which is why we start WEBrick using the `-e` switch.

Next, make sure that the `forum_posts` table in your test database is empty by executing the following command and SQL:

```
$ mysql –uemporium –phacked emporium_test
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 74 to server version: 5.0.20-community

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql> delete from forum_posts;
```

```
Query OK, 13 rows affected (0.09 sec)
```

You can also tell Selenium to empty the table automatically for you at the start of the test, as we'll show you later in this section.

Now open Firefox and start Selenium IDE by selecting Tools ➤ Selenium IDE from the main menu in the browser. You should see the Selenium IDE (see Figure 11-4).

■ **Tip** To be able to record tests in Selenese, you have to add a new custom test format. This is done by clicking Add on the Formats tab in the options dialog. The source for the Selenese "wiki-like" format can be downloaded from this page: `http://wiki.openqa.org/display/SIDE/SeleniumOnRails`.

Select the Selenese output format from the Options ➤ Format menu in the Selenium IDE window. This makes Selenium IDE generate the acceptance tests in Selenese instead of the default HTML. When started, Selenium IDE is in record mode by default, so you can start recording immediately.

Follow these steps to record the test:

1. Open `http://localhost:3000/forum`. You should now see the forum main page showing the text "There are no posts yet."
2. Right-click somewhere on the page and select Show All Available Commands ➤ assertTitle Forum from the pop-up menu.
3. Select the text "There are no posts yet."
4. Right-click the selected text and select `verifyTextPresent There are no posts yet` from the pop-up menu.
5. Right-click the "Be the first one to post here" link, then select `assertText link=Be the first one to post here` from the pop-up menu. This verifies that the page has a link that says "Be the first one to post here."
6. You have now recorded the first two requirements of the acceptance test, and the Selenium IDE Table tab should look like Figure 11-5. Save the test by selecting File ➤ Save Test from the Selenium IDE menu.

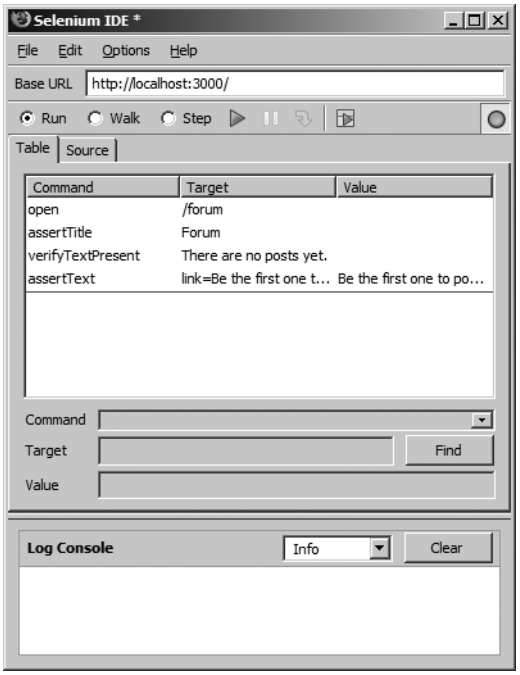


Figure 11-5. *The first version of the acceptance test*

■ **Tip** You can execute the acceptance test and view the results directly in Selenium IDE by clicking the green play button in the Table tab. You can also watch the test being executed in slow-motion by selecting the Walk mode. Use the Step mode to step through each command in the test manually.

Next, we need to verify that pagination works correctly when there are more than 20 posts in the forum. To prepopulate the database with test data, we can use the following fixture:

```
<% 40.times do |i| %>post_<%= i %>:
  id: <%= i %>
  name: Post <%= i %> name
  subject: Post <%= i %> subject
  body: Post <%= i %> body
  created_at: <%= Time.now.to_s(:db) %>
  updated_at: <%= Time.now.to_s(:db) %>
<% end %>
```

Save the fixture in `test/fixtures/forum_posts.yml`. The dynamic fixture adds 40 posts to the database, which means there should be two pages in the forum, and you should see Next page and Previous page links when navigating between the pages.

Selenium on Rails supports fixtures in the same way as the built-in Rails unit, integration, and functional tests. Use the `open` command to load a special URL that inserts all the specified fixtures into the database. Load multiple fixtures by separating them with commas:

```
|open|/selenium/setup?fixtures=beer,wine,booze||
```

Now add a new command to the test by selecting the line after the last `assertText` command in the list. Type `open` in the Command field, and enter `/selenium/setup?fixtures=forum_posts` in the Target field, as shown in Figure 11-6.

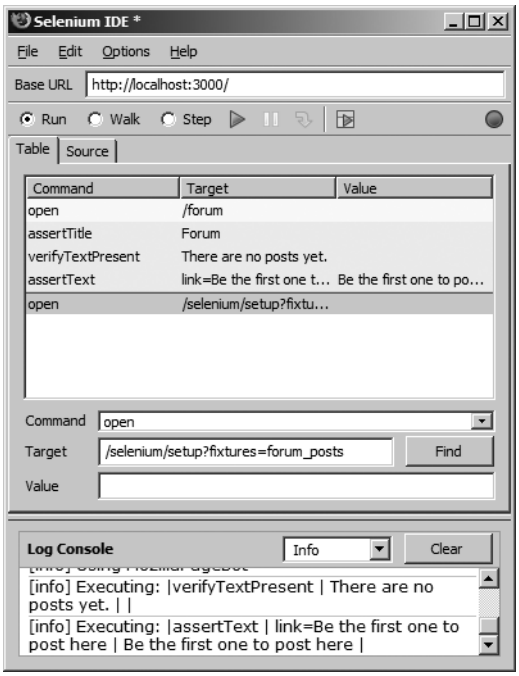


Figure 11-6. *Adding a fixture to the test*

Now that we are inserting data in the tests, the test will fail if we execute it twice in succession, so we need to make sure that the `forum_test` database table is empty at the start of the test. We do this by adding the following to the start of the script:

```
|open|/selenium/setup?clear_tables=forum_posts||
```

The trick to emptying tables of all data is to use the `clear_tables` parameter when calling the `setup` action. To empty more than one table, separate the table names with commas.

We also want to verify that the list is showing exactly 20 posts. We do this with the `verifyElementNotPresent` command and an XPath element locator that tries to find the twenty-first post, which it shouldn't find on the page:

```
|verifyElementNotPresent|//div[@id='posts']/div[21]||
```

We can test that the navigation between the two pages works by adding the following commands:

```
|clickAndWait|link=Next page||
|clickAndWait|link=Previous page||
```

After adding these commands, the test should look as follows (you can verify this by clicking the Source tab, as shown in Figure 11-7):

```
|open|/selenium/setup?clear_tables=forum_posts||
|open|/forum||
|assertTitle|Forum||
|verifyTextPresent|There are no posts yet.||
|assertText|link=Be the first one to post here|Be the first one to post here|
|open|/selenium/setup?fixtures=forum_posts||
|verifyElementNotPresent|//div[@id='posts']/div[21]||
|clickAndWait|link=Next page||
|clickAndWait|link=Previous page||
```

Figure 11-7. *The source for the View Forum acceptance test shown in Selenium IDE*

Save the changes, and then run the test once by clicking the green run button in Selenium IDE. We can now execute the acceptance tests with `rake`:

```
$ rake test:acceptance
```

The test should pass, as shown in Figure 11-8.

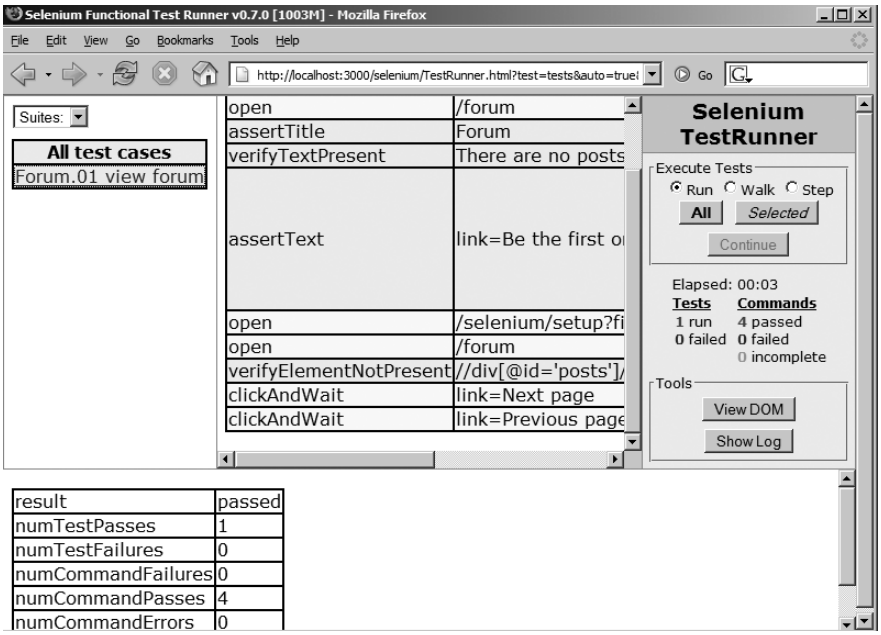


Figure 11-8. *The results after running the final version of the View Forum acceptance test*

Recording the Post to Forum Acceptance Test

In the acceptance test for the Post to Forum user story, which describes how a post is created, we want to verify that posting to the forum works and that the post is shown on the forum main page.

Record the acceptance test by following these steps:

1. Select File ➤ New Test from the Selenium IDE menu. Verify that Selenium IDE is recording, by checking that the red record button is activated.

2. Open `http://localhost:3000/forum` in your browser. Line 1 should now contain an `open` command. You should also see the posts that were created by the previous test case.

3. Click the New post link at the top of the page. You should see a `clickAndWait` command inserted at line 2.

4. On the new post page, type a name (for example, Luke Rhinehart) in the Name field.
5. In the Subject field, type a post subject (for example, The Dice Man).
6. In the Body field, type a message (for example, Roll the die!).
7. Click the Post button. You should be redirected to the main page of the forum, where you should see the post.
8. Select the text ("The Dice Man by Luke Rhinehart" in our example).
9. Right-click the selected text and select `verifyTextPresent The Dice Man by Luke Rhinehart` from the pop-up menu.
10. Save the recorded test as `test/selenium/forum/02_post_to_forum.sel` (by selecting File ➤ Save Test from the Selenium IDE menu).

Run the test by clicking the run button in Selenium IDE. You should see the test succeed (all rows are green), as shown in Figure 11-9.

Figure 11-9. *The Post to Forum acceptance test in Selenium IDE*

If you open `02_post_to_forum.sel` in an editor, you should see something like the following:

```
|open|/forum||
|clickAndWait|link=New post||
|type|post[name]||Luke Rhinehart|
|type|post[subject]||The Dice Man|
|type|post[body]||Roll the die!|
|clickAndWait|commit||
|verifyTextPresent|The Dice Man by Luke Rhinehart||
```

Now test that the View Forum and Post to Forum acceptance tests work when both are run in succession by executing the following:

```
$ rake test:acceptance
```

Recording the Show Post Acceptance Test

Next, we'll create a test that verifies that the Show Post user story works as intended. You should now have the forum main page open in Firefox, which is where we left off in the previous section.

Like the other tests, the test case requires that we are on the page where the previous test ended. We'll use the `assertLocation` check to verify this. Record the acceptance test by following these steps:

1. Select File ➤ New Test from the Selenium IDE menu. Verify that Selenium IDE is recording, by checking that the red record button is activated.
2. Click the Source tab and type `|assertLocation|/forum||` in the text area, as shown in Figure 11-10. Switch back to the previous view by clicking the Table tab.

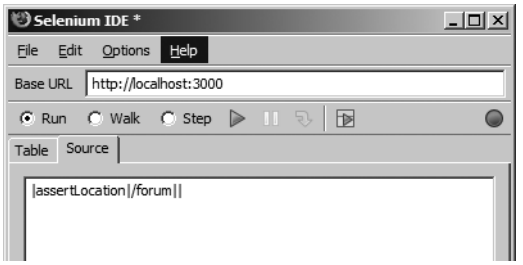


Figure 11-10. *The Source tab showing the acceptance test after step 2*

3. Click the second line in the table.
4. Click the post that was created by the Post to Forum acceptance test (The Dice Man). You are now taken to the show post page.
5. Select the text "The Dice Man," right-click it, and select `verifyTextPresent The Dice Man` from the pop-up menu.
6. Select the text "Luke Rhinehart," right-click it, and select `verifyTextPresent Luke Rhinehart` from the pop-up menu.
7. Right-click the Reply link and select `assertTextLink link=Reply Reply` from the pop-up menu.
8. Save the test as `test/selenium/forum/03_show_post.sel` (by selecting File ➤ Save Test from the Selenium IDE menu).

The file should now contain the following:

```
|assertLocation|/forum||
|clickAndWait|link=The Dice Man||
|verifyTextPresent|'The Dice Man'||
|assertText|link=Reply|Reply|
```

Verify that what you have done so far works by executing the acceptance tests once again:

```
$ rake test:acceptance
```

You should see the acceptance tests run without failures.

Recording the Reply to Post Acceptance Test

The Reply to Post acceptance test is run after the Show Post test, and it is used to verify that replying to the post works. The Reply to Post user story is similar to the Post to Forum user story. Record it by following these steps:

1. Select File ➤ New Test from the Selenium IDE menu. Verify that Selenium IDE is recording, by checking that the red record button is activated.
2. We want to verify that we are on the page where the previous acceptance test left us. Right-click somewhere on the page and choose `assertTitle 'The Dice Man'` from the pop-up menu.

■ **Note** If step 2 adds an `open` command to the acceptance test, remove it from the list. You need to get rid of it because it contains the unique `id` of the post, which won't be the same when you run the test again.

3. Click the Reply link. You should now see the reply to post page.
4. Type a name (for example, George) in the Name field.
5. Type a subject (for example, Let's play) in the Subject field.
6. Type a message (for example, I'm starting today!) in the Body field.
7. Click the Reply button. You are taken to the forum main page.
8. Select the text ("Let's play by George" in our example), right-click it, and select `verifyTextPresent Let's play by George` from the pop-up menu.
9. Switch to the Source view by clicking the Source tab. Add the following to the end of the test: `|assertLocation|/forum||`.
10. Save the test as `test/selenium/forum/04_reply_to_post.sel` (by selecting File ➤ Save Test from the menu).

The test should now contain the text shown here:

```
|assertTitle|'The Dice Man'||
|clickAndWait|link=Reply||
|type|post[name]|George|
|type|post[subject]|Let's play|
|type|post[body]|I'm starting today!|
|clickAndWait|commit||
|verifyTextPresent|Let's play by George||
|assertLocation|/forum||
```

We have now created acceptance tests for all of the forum-related user stories. Let's show George the wonder of automation by executing the complete set of Selenium acceptance tests:

```
$ rake test:acceptance
```

Firefox starts up, and Selenium then executing our acceptance tests one by one. Figure 11-11 shows the result page, which appears after the tests have run. All tests should be green, which indicates that they passed.

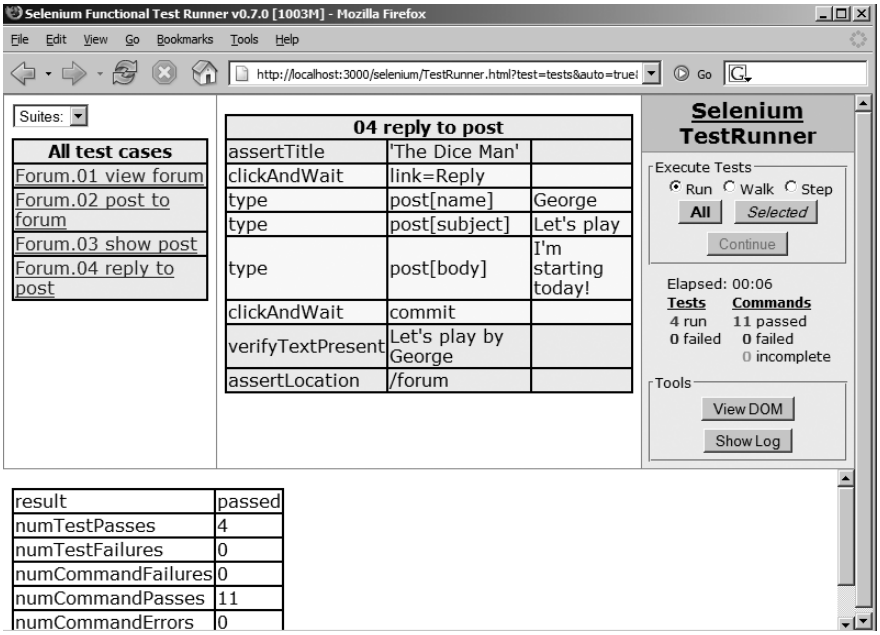


Figure 11-11. *Running the recorded acceptance tests*

Next, click the Walk radio button in the Execute Tests section of TestRunner window, and then click the All button beneath it to run the tests again. Selenium now runs the tests in a slow-motion mode that allows you to see each command being executed.

You can also try stepping through the tests, by clicking the Step radio button in the Execute Tests section and clicking the All button. Instead of executing the whole test, Selenium stops after each executed command. To continue the test, click the Continue button. This is useful when you're debugging a test.

Summary

In this chapter, you learned how to write automated acceptance tests using Selenium and the Selenium on Rails plugin. By automating acceptance tests, you not only remove manual work, but you also raise your confidence that your application works according to the requirements. We also showed you how to simplify the process of writing acceptance tests by using the Selenium IDE Firefox extension to record the tests.

In the next chapter, we'll show you how to deploy your application to production. There shouldn't be any surprises, as we have implemented a full set of automated tests that exercise almost all parts of the application.

CHAPTER 12

Application Deployment

In this chapter, we'll show you how to set up an application's production environment, including the LightTPD web server and FastCGI extension. Then we will walk through the manual deployment process. Finally, we'll demonstrate how to simplify deployment tasks with Capistrano, a tool specifically designed to automate the deployment of Ruby on Rails applications.

Setting Up the Production Environment

George has bought a new Intel-powered server running Ubuntu Linux, to which we will deploy LightTPD (web server), Ruby on Rails and FastCGI (application server), and MySQL (database server). The high-level system architecture of the Emporium production environment is shown in Figure 12-1.

Although we talk about three different servers, the production environment consists of only one physical machine, since that is all we need to start. Later, we can support more traffic if the need arises by scaling horizontally (adding more machines) or vertically (adding more processing power).

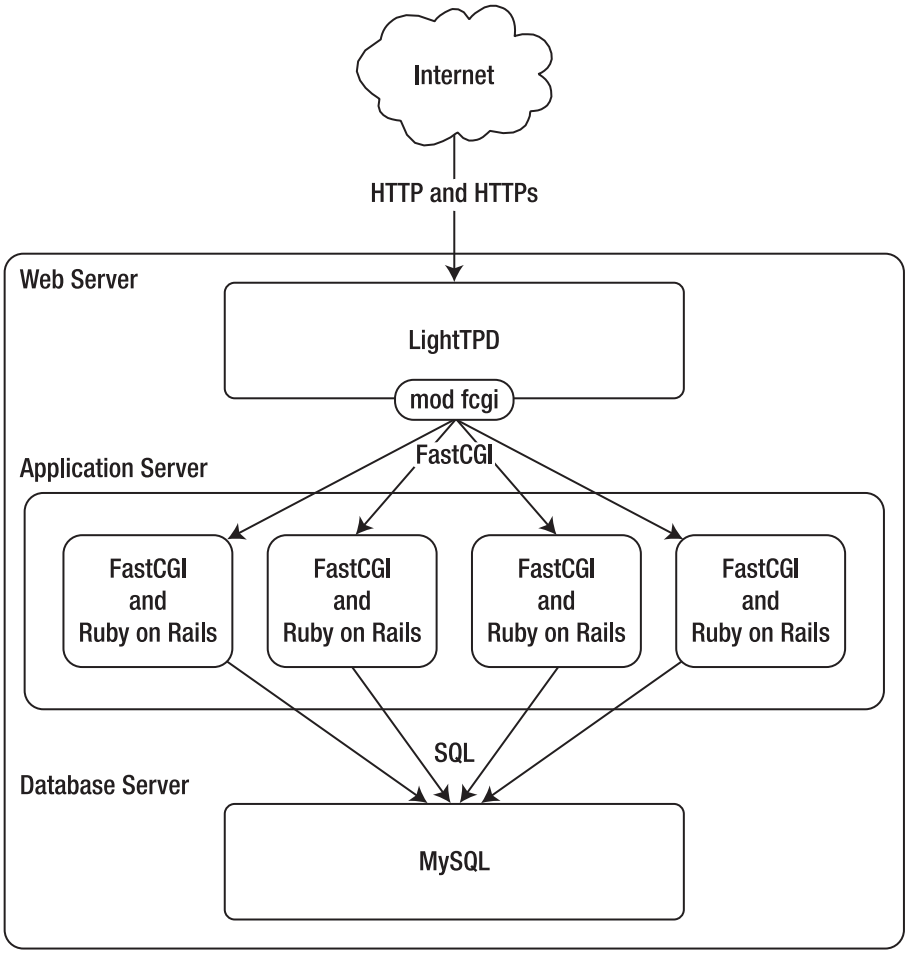


Figure 12-1. *The Emporium production environment*

Connecting to the Production Server: SSH

We will be using Secure Shell (SSH) throughout this chapter to connect to the production machine. If you haven't installed the SSH server on your production machine yet, you need to do it now before proceeding. Log in to the production server and execute the following command:

```
$ sudo apt-get install openssh-server
```

After the installation is complete, you can start the SSH server with this command:

```
$ sudo /etc/init.d/ssh start
```

```
* Starting OpenBSD Secure Shell server…
```

With the SSH server running on the production machine, connect to the production server by executing the following:

```
$ ssh username@production_server_ip
```

■ **Note** Originally developed by Tatu Ylönen, SSH is a set of standards and protocols that allow you to establish a secure connection to a remote server. We assume that you have a basic understanding of what SSH is and how it can be used. For more information about SSH, see Wikipedia's entry on SSH (`http://en.wikipedia.org/wiki/Secure_Shell`).

You are now ready to start installing the software on your production environment.

Installing the Web Server: LightTPD

The web server handles the communication between the user's browser and the Emporium application, which is running on the application server in one or more separate processes. When a request comes in from the Internet, the web server forwards the request (by acting as a reverse proxy and a load balancer) to one of the application servers.

In this book, we use LightTPD as the web server, because it is a tried-and-tested open source web server that, according to its homepage (`http://www.lighttpd.net`) is optimized for high performance. This claim is backed up by benchmarks, which put it among the fastest web servers available currently, and in front of the more popular Apache in some areas. LightTPD is easy to configure and includes the `mod_fastcgi` module, which can communicate with external FastCGI processes running Ruby on Rails applications. After being forwarded by LightTPD, the request is processed by Ruby on Rails, and the output is sent back to the client, again using the FastCGI protocol.

■ **Tip** The LightTPD wiki (`http://trac.lighttpd.net/trac/wiki/`) is a good starting point if you want to find out more about LightTPD and how to use it with Ruby on Rails.

Here, we provide instructions for installing LightTPD from source on Ubuntu Linux. You can also find a binary installation package for Windows, Mac OS X, and most Linux distributions at `http://www.lighttpd.net/download`.

Although some commands are specific to Ubuntu Linux, the installation instructions are generally applicable to most Linux/Unix platforms, with some exceptions, such as how dependencies are installed. See the Rails wiki (`wiki.rubyonrails.org`) and other online resources for detailed instructions on how to set up and configure the software on other platforms.

■ **Tip** If you are using Mac OS X, you can find a good resource for how to set up Rails, LightTPD, FastCGI, and MySQL at `http://hivelogic.com/articles/2005/12/01/ruby_rails_lighttpd_mysql_tiger`.

LightTPD has the following external dependencies, which must be installed separately and before LightTPD itself:

- The Perl Compatible Regular Expressions (PCRE) library, used by `mod_rewrite` and other modules
- The `zlib` compression library, used for enabling `gzip` and other compression support
- OpenSSL for enabling HTTPS and Secure Sockets Layer (SSL)

On Ubuntu Linux, you can install the dependencies with the `apt-get` command:

```
$ sudo apt-get install libpcre3-dev
$ sudo apt-get install zlib1g-dev
$ sudo apt-get install open-ssl
$ sudo apt-get install libssl-dev
```

You also need to install the compiler provided by GCC, which is available through the `build-essentials` package in Ubuntu Linux:

```
$ sudo apt-get install build-essentials
```

Next, download the latest available stable source LightTPD package from `http://www.lighttpd.net/download` to a directory on your production server (such as `/usr/local/src` or `/tmp`). Then execute the following commands one by one (we've omitted the output of the commands here for the sake of clarity):

```
$ tar xzvf lighttpd-1.x.x.tar.gz
$ cd lighttpd-1.x.x
$ ./configure --with-openssl
$ make
$ sudo make install
```

These commands extract the package and compile the source.

■ **Tip** For more information about compiling LightTPD, refer to the `INSTALL` and `README` files, which are located inside the source package.

As we said, you need to install both the OpenSSL package and the SSL development libraries and headers in order to run LightTPD. If you didn't install them, you would get an error similar to the one shown here when starting LightTPD:

```
SSL: ssl requested but openssl support is not compiled in
```

You can verify that the required dependencies are found by inspecting the output of the `configure` command:

```
.
checking for OpenSSL... yes
checking openssl/ssl.h usability... yes
checking openssl/ssl.h presence... yes
.
.
checking pcre.h usability... yes
checking pcre.h presence... yes
checking for pcre.h... yes
checking for deflate in -lz... yes
checking zlib.h usability... yes
checking zlib.h presence... yes
checking for zlib.h... yes
.
```

If you don't see any errors while compiling and installing the source, it means you can continue and verify that the installation was successful. Execute the following command to print the version information of the binary you just compiled:

```
$ lighttpd -v
```

```
lighttpd-1.4.11 - a light and fast webserver
Build-Date: May 14 2006 20:46:11
```

We'll show you how to configure LightTPD after we install the other software.

Installing the Application Server: Ruby on Rails and FastCGI

The application server is where Emporium is running inside one or more Ruby processes that use the FastCGI protocol to communicate with the web server. These processes, also referred to as *dispatchers*, listen for incoming requests, process the request, and then send the response back to the web server. The processes are then made available for processing the next request.

FastCGI (`http://www.fastcgi.com/`) is an open extension to CGI. The biggest problem with CGI comes from the fact that each request to the web server starts a new process, and each process requires some startup and cleanup tasks to be performed. As its name implies, FastCGI removes most of the performance and scalability problems associated with CGI by using a pool of long-running processes.

FastCGI also promises security enhancements. Multiple load-balanced FastCGI processes can run on remote machines instead of locally on the web server like CGI. This means that you can run the FastCGI processes and the web server under different users. Then a hacker trying to gain access to your system must hack into both accounts: the one running your web server and the one running the FastCGI processes.

To install the FastCGI library, download the source for the latest stable version from `http://www.fastcgi.com/dist/`, and then compile it according to the installation instructions found in the `INSTALL` and `README` files, which are located in the root of the distribution package:

```
$ tar zxvf fcgi-x.x.x.tar.gz
$ cd fcgi-x.x.x
$ ./configure
$ make
$ sudo make install
```

■ **Note** FastCGI can also be installed through a binary distribution on most platforms. However, installing from source usually works better.

You also need to install the Ruby-FastCGI library to allow your Ruby on Rails application to communicate with the web server. First, download the latest version of the Ruby-FastCGI library from `http://raa.ruby-lang.org/project/fcgi`. Then change the current directory to where you downloaded the package and execute the following commands:

```
$ tar zxvf ruby-fcgi-x.x.x.tar.gz
$ cd ruby-fcgi-x.x.x
$ ruby install.rb config
$ ruby install.rb setup
$ sudo ruby install.rb install
```

If you are having problems installing the library, check the README file. Some systems might require that you specify where the FastCGI headers and library files are located. You can do this by executing the install script with the following parameters:

```
ruby install.rb config -- --with-fcgi-include=/usr/local/include ➥
--with-fcgi-lib=/usr/local/lib
```

■ **Note** The Ruby-FCGI library can also be installed through the RubyGems packaging system by executing sudo gem install fcgi.

The Ruby-FastCGI library contains two different implementations: one native implementation written in C and one written in pure Ruby. In a production environment, you want to use the native implementation because of the performance benefits it provides. By default, the native implementation will be used, but only if the FastCGI shared library can be found. This shared library is created when you compile and install FastCGI from source. To verify that Ruby-FastCGI uses the native implementation of FastCGI, execute the following commands in the interactive Ruby console, irb:

```
$ irb
require 'fcgi.so'
```

```
=> true
```

The line require 'fcgi.so' should return true, as shown here. If it returns false, it means it cannot find the fcgi.so library, and so it will run using the pure Ruby implementation.

If you get an error saying fcgi.so cannot be found, you might need to add the path to /usr/local/lib to /etc/ld.so.conf and run ldconfig.

You can verify that the pure Ruby (slower) version is found by executing the following in irb:

```
$ irb
> require 'fcgi'
```

```
=> true
```

ALTERNATIVES TO LIGHTTPD, FCGI, AND MYSQL

You have a multitude of options to choose from when planning your production environment. Most often, the best strategy is to follow Ruby on Rails best practices and to concentrate on keeping the architecture simple.

Running your application on LightTPD is a safe option, as it is used by many existing Ruby on Rails applications, and can be considered to be fairly simple to install and maintain. But LightTPD is by no means the only option. Apache (`httpd.apache.org`) might be a better option for some applications and platforms; for example, Basecamp (`www.basecamphq.com`) runs on Apache. Other alternatives include any web server that supports FastCGI or that can act as a proxy for Mongrel.

Mongrel (`mongrel.rubyforge.org`), an alternative to using FastCGI, is a fast HTTP library and server that is slowly becoming the de facto standard for new Rails production deployments. With Mongrel, there's no need for FastCGI, because Mongrel itself talks HTTP and acts as a web server. This simplifies the deployment and maintenance of applications. Note that at the time of writing, the author of Mongrel, Zed Shaw, recommends using Apache, rather than LightTPD, with Mongrel, because of problems with its `mod_proxy` module. This will probably be fixed when the new `mod_proxy_core` module is released. See Coda Hale's blog post (`http://blog.codahale.com/2006/06/19/time-for-a-grown-up-server-rails-mongrel-apache-capistrano-and-you/`) for a write-up on Mongrel, and how to use it together with Apache and Capistrano.

Ruby on Rails plays well with most of the popular database servers found on the market today, both open source and commercial. If you're looking for an open source database server similar to MySQL, we recommend PostgreSQL (`www.postgresql.org`).

Installing the Database Server (MySQL)

In the case of Emporium, the database server is running a single instance of MySQL (`http://www.mysql.com`). MySQL is claimed to be the world's most popular open source database. It is a safe choice that is used by many high-traffic websites, including craigslist.com, which serves millions of classified ads a day from a MySQL cluster.

The communication between Ruby on Rails and the database is done through a native MySQL database driver.

You need to install MySQL and the MySQL driver (native) on the production server. We explained how to install these in Chapter 1.

Configuring LightTPD

Configuring LightTPD is straightforward. You can use the template (`doc/lighttpd.conf`) that is distributed along with the source and customize it to meet your needs.

The configuration file for Emporium's production environment is shown in Listing 12-1. Save this configuration in `$EMPORIUM_PATH/config/lighttpd_production.conf`, where `$EMPORIUM_PATH` is the path to your application.

Listing 12-1. *LightTPD Configuration File*

```
server.modules = (
  "mod_rewrite",
  "mod_access",
  "mod_fastcgi",
  "mod_compress",
  "mod_accesslog" )

# Deny access to potentially dangerous files
url.access-deny             = ( "~", ".inc" )

# Deny access to URLs matching the specified regexp
# In this case subversion files
$HTTP["url"] =~ "/\.svn/" {
  url.access-deny = ( "" )
}

# Enable HTTPS/SSL
$SERVER["socket"] == "0.0.0.0:443" {
    ssl.engine = "enable"
    ssl.pemfile = "/u/apps/emporium/current/config/server.pem"
}

# Listen on all network interfaces on port 80
server.port         = 80
server.bind         = "0.0.0.0"
server.pid-file     = "/var/run/lighttpd.pid"

server.document-root        = "/u/apps/emporium/current/public"
server.indexfiles           = ( "index.html", "dispatch.fcgi" )
server.error-handler-404    = "/dispatch.fcgi"
server.errorlog             = "/var/log/lighttpd/lighttpd_error.log"
server.tag                  = "Emporium Server 1.0"

server.username            = "lighttpd"
server.groupname           = "lighttpd"

accesslog.filename          = "/var/log/lighttpd/lighttpd_access.log"

# Four FastCGI processes running locally
fastcgi.server = ( ".fcgi" =>
    ( "emporium-7000" => ( "host" => "127.0.0.1", "port" => 7000 ) ),
    ( "emporium-7001" => ( "host" => "127.0.0.1", "port" => 7001 ) ),
    ( "emporium-7002" => ( "host" => "127.0.0.1", "port" => 7002 ) ),
    ( "emporium-7002" => ( "host" => "127.0.0.1", "port" => 7002 ) )
)
include "mimetypes.conf"
```

The syntax of the configuration file uses the following format:

```
option = value
```

Options are usually grouped into modules, such as `server.port` and `accesslog.filename`. Supported value formats include strings, integers, booleans, arrays, and others.

■ **Note** For more information about the LightTPD configuration file, see the wiki entries on the supported configuration syntax and options, located at `http://trac.lighttpd.net/trac/wiki/Docs%3AConfiguration` and `http://trac.lighttpd.net/trac/wiki/Docs%3AConfigurationOptions`, respectively.

Let's have a closer look at each section of the configuration file.

Module Configuration

The first line in Listing 12-1, `server.modules`, tells LightTPD which modules it should load and enable at startup. Similar to Apache modules, LightTPD modules extend the base functionality of LightTPD. In Listing 12-1, we enabled the following modules:

- `mod_rewrite`: Allows you to rewrite, or modify, the URL that was used to access a server resource. For example, you could configure `mod_rewrite` so that requests for URLs ending with `.html` are stripped of the `.html` suffix and processed by Ruby on Rails. For example, `/catalog/index.html` is rewritten to `/catalog/index`.
- `mod_access`: Allows you to deny access to certain files that match a regular expression pattern.
- `mod_fastcgi`: Used to communicate with external FastCGI processes.
- `mod_compress`: Adds support for `deflate`, `gzip2`, and `bzip2` content compression. Enabling content compression allows you to save bandwidth and makes it faster for the browser to download the content.
- `mod_accesslog`: Logs each request to a file or other supported destination.

Each module is configured separately and adds a set of options that can be used in the configuration file.

■ **Note** See the module documentation for more information. You can find links for each module's documentation at `http://trac.lighttpd.net/trac/wiki/Docs`.

Log File Configuration

LightTPD maintains an access log, an error log, and a PID file, which all tell you something about the server's status. Most system and application logs can be found in the `/var/log` directory, so a good place to keep the access and error logs is `/var/log/lighttpd/`. For LightTPD to be able to write to these directories, you must first create the directories, and then change the access rights, as follows:

```
$ sudo mkdir /var/log/lighttpd
$ sudo chgrp lighttpd /var/log/lighttpd
$ sudo chmod g+rw /var/log/lighttpd
```

The access log is where LightTPD keeps a log of all requests that have been made to the server. Among other things, the access log is useful for generating usage reports with, for example, AWStats (`http://awstats.sourceforge.net/`) and Webalizer (`http://www.mrunix.net/webalizer/`). You tell LightTPD where to write the access log data with the `accesslog.filename` setting in the configuration file (Listing 12-1). A typical entry in the access log might look like this:

```
127.0.0.1 www.emporium.com - [13/Aug/2006:07:45:45 +0000] "GET / ...
```

The LightTPD error log is specified with the `server.errorlog` setting in the configuration file. The following are typical entries in this log:

```
2006-06-06 09:57:50: (log.c.133) server stopped
2006-06-06 09:58:19: (log.c.75) server started
```

The location of the PID file is specified with the `server.pid-file` setting in the configuration file. The PID file is created by LightTPD at startup, and it contains the process ID of LightTPD. The file should be located in the `/var/run` directory, along with PID files of other processes.

Mime-Type Configuration

The last line in the configuration file (Listing 12-1), `include "mimetypes.conf"`, tells LightTPD to include the mime-type configuration file. This configures LightTPD so that it responds with the correct `Content-Type` HTTP header matching the requested file. For example, the web server sends the header `Content-Type: application/pdf` when a browser requests a PDF file with the extension `.pdf`.

Save the configuration shown in Listing 12-2 in `config/mimetypes.conf`.

Listing 12-2. *Mime-Type Configuration*

```
mimetype.assign             = (
  ".xpi"          =>      "application/x-xpinstall",
  ".rdf"          =>      "application/xml",
  ".xul"          =>      "application/vnd.mozilla.xul+xml",
  ".pdf"          =>      "application/pdf",
  ".sig"          =>      "application/pgp-signature",
  ".spl"          =>      "application/futuresplash",
  ".class"        =>      "application/octet-stream",
  ".ps"           =>      "application/postscript",
  ".torrent"      =>      "application/x-bittorrent",
  ".dvi"          =>      "application/x-dvi",
  ".gz"           =>      "application/x-gzip",
  ".pac"          =>      "application/x-ns-proxy-autoconfig",
  ".swf"          =>      "application/x-shockwave-flash",
  ".tar.gz"       =>      "application/x-tgz",
  ".tgz"          =>      "application/x-tgz",
  ".tar"          =>      "application/x-tar",
  ".zip"          =>      "application/zip",
  ".mp3"          =>      "audio/mpeg",
  ".m3u"          =>      "audio/x-mpegurl",
  ".wma"          =>      "audio/x-ms-wma",
  ".wax"          =>      "audio/x-ms-wax",
  ".ogg"          =>      "audio/x-wav",
  ".wav"          =>      "audio/x-wav",
  ".gif"          =>      "image/gif",
  ".jpg"          =>      "image/jpeg",
  ".jpeg"         =>      "image/jpeg",
  ".png"          =>      "image/png",
  ".xbm"          =>      "image/x-xbitmap",
  ".xpm"          =>      "image/x-xpixmap",
  ".xwd"          =>      "image/x-xwindowdump",
  ".css"          =>      "text/css",
  ".html"         =>      "text/html",
  ".htm"          =>      "text/html",
  ".js"           =>      "text/javascript",
  ".asc"          =>      "text/plain",
  ".c"            =>      "text/plain",
  ".conf"         =>      "text/plain",
  ".text"         =>      "text/plain",
  ".txt"          =>      "text/plain",
  ".dtd"          =>      "text/xml",
  ".xml"          =>      "text/xml",
  ".mpeg"         =>      "video/mpeg",
  ".mpg"          =>      "video/mpeg",
  ".mov"          =>      "video/quicktime",
```

```
  ".qt"           =>      "video/quicktime",
  ".avi"          =>      "video/x-msvideo",
  ".asf"          =>      "video/x-ms-asf",
  ".asx"          =>      "video/x-ms-asf",
  ".wmv"          =>      "video/x-ms-wmv"
)
```

■ **Note** You can also use the mime-type listing found in LightTPD's configuration template (`doc/lighttpd.conf`).

Access Configuration

Letting your web server blindly serve all files will most likely cause security problems in a production environment. Your web server might serve files containing sensitive information like backups created by vi and emacs or files used by Subversion. To deny access to these files, the configuration file (Listing 12-1) defines two rules using `url.access-deny`: one for backups, as defined in the LightTPD template, and one for Subversion files.

Later in this chapter, we will use Capistrano to deploy the application to production. By default, Capistrano uses the Subversion `checkout` command when deploying the application to the production machine. Using the Subversion `checkout` command, instead of the `export` command, means that the deployment directory will contain `.svn` directories, which could be served by your web server, if someone is smart enough to request them. Here is an example of the information that can be found in `.svn/entries`:

```
<?xml version="1.0" encoding="utf-8"?>
<wc-entries
   xmlns="svn:">
<entry
   committed-rev="106"
   name=""
   committed-date="2006-04-11T21:07:20.659809Z"
   url="svn://127.0.0.1:3690/emporium/public"
   last-author="george"
   kind="dir"
   uuid="1612fdca-df0d-0410-9dbd-93b5c6b9c7f0"
   prop-time="2006-04-19T20:19:36.000000Z"
   revision="109"/>
```

As highlighted in the example, a hacker can find out the URL of your Subversion server and the user that updated the file.

You can prevent access to all files and folders named `.svn` using the following rule in the `lighttpd` configuration file (as described in `http://hivelogic.com/articles/2006/04/30/preventing_svn_exposure`):

```
$HTTP["url"] =~ "/\.svn/" {
  url.access-deny = ( "" )
}
```

SSL Configuration

The communication between a browser and an e-commerce site needs to be secured through encryption to prevent theft of sensitive information like credit card numbers and login credentials. The SSL protocol is the de facto standard for secure communication on the Internet. SSL uses public-key encryption and requires that you acquire an SSL certificate from a certified issuer like VeriSign (`www.verisign.com`) or Thawte (`www.thawte.com`). To apply for an SSL certificate, go to the issuer's website and select the appropriate SSL certificate.

The part of the configuration file (Listing 12-2) that enables SSL is shown here:

```
# Enable HTTPS/SSL
$SERVER["socket"] == "0.0.0.0:443" {
    ssl.engine = "enable"
    ssl.pemfile = "/u/apps/emporium/current/config/server.pem"
}
```

The IP and port is specified with `$SERVER["socket"]`. The HTTPS port should always be 443. Note that specifying `0.0.0.0` configures LightTPD to listen to all network interfaces, which might not be desired. Instead, you could set it to the public IP of your server. The `ssl.pemfile` configuration property should point to your SSL certificate file that you received from the issuer.

Before starting LightTPD, you need to acquire the SSL certificate or remove the SSL part from the configuration; otherwise, you will get an error when you try to start LightTPD.

■ **Tip** If you don't want to buy a certificate immediately, you can generate a self-signed SSL certificate, which is valid for 365 days, with this OpenSSL command: `openssl req -new -x509 -keyout server.pem -out server.pem -days 365 –nodes`. A self-signed certificate is not very useful in a production environment, because users will receive a warning when accessing your site, saying that the certificate was not created by a trusted issuer. However, such a certificate is handy for development and testing purposes.

FastCGI Module Configuration

In the configuration file (Listing 12-1), the FastCGI module is configured to proxy requests to four FastCGI processes running on different ports on the same machine as the web server, as shown here:

```
fastcgi.server = ( ".fcgi" =>
    ( "emporium-7000" => ( "host" => "127.0.0.1", "port" => 7000 ) ),
    ( "emporium-7001" => ( "host" => "127.0.0.1", "port" => 7001 ) ),
    ( "emporium-7002" => ( "host" => "127.0.0.1", "port" => 7002 ) ),
    ( "emporium-7002" => ( "host" => "127.0.0.1", "port" => 7002 ) )
)
```

Scaling horizontally, by adding more application servers to your environment, is easy with FastCGI—just install the new machines and add them to the list.

The FastCGI processes are started by the `spawner` script located in the `script/process` directory. We will show you how to manage FastCGI processes later in this chapter, in the sections about manual and automated deployment.

Creating the Production Database

Before deploying and starting the application, you need to create the production database. Without it, your application wouldn't work and Rails wouldn't even start in production mode. Log in to the remote server and execute the following commands:

```
$ mysql -uroot
create database emporium_production;
grant select,insert,update,delete,create,drop on ➥
emporium_production.* to 'emporium'@'localhost' identified by 'hacked';
flush privileges;
```

This creates the `emporium_production` database and the MySQL user that is used when connecting to the database. Notice that we are not granting all rights to the user, as was the case with the development and test databases that we created in earlier chapters. Instead, we are granting only the minimum privileges required by the application: `select`, `insert`, `update`, `delete`, `create`, and `drop`. It's not a good idea to give `grant` and `show` database privileges, for example, since they could be used by a hacker to gain access to other databases.

Next, update the database configuration file (`config/database.yml`) as shown here:

```
development:
  adapter: mysql
  database: emporium_development
  username: emporium
  password: hacked
  encoding: utf8
test:
  adapter: mysql
  database: emporium_test
  username: emporium
  password: hacked
  encoding: utf8
production:
  adapter: mysql
  database: emporium_test
  username: emporium
  password: hacked
  encoding: utf8
```

Finally, run the migrations by executing the migration scripts:

```
rake db:migrate RAILS_ENV=production
```

We are now ready to deploy the application to production. We'll first describe how to deploy it manually, so that you can appreciate the benefits of using Capistrano to automate the deployment process. It's also good to do a manual deployment first to verify that everything is set up and configured correctly.

Deploying the Application Manually

Deploying an application to your production server manually requires the following steps:

- Copy the application to the production environment.
- Create users and groups for the owners of the LightTPD, FastCGI, and `spawner` processes.
- Start LightTPD.
- Start the FastCGI processes.

As you'll see, the procedure isn't difficult, but there are many details to handle, and it is error-prone.

Copying the Application

First, you need to copy the application to the production environment. We are assuming that you have been using Subversion during the development of the Emporium application. This means the source code is located in the Subversion repository, and that you can deploy the code to production by using the Subversion `checkout` command:

```
$ svn co svn://localhost/emporium/trunk /u/apps/emporium/current/
```

This checks out the latest version of your project to `/u/apps/emporium/current/`.

If you don't have Subversion installed on the production machine, you can execute the following command to install it:

```
$ sudo apt-get install subversion
```

With Subversion installed, you can create a new repository and import the source or copy over your previous repository. Then start the Subversion server with the following command:

```
$ svnserve -d -r /home/george/subversion/repository --listen-host 127.0.0.1
```

This starts the `svnserve` daemon on the local machine and uses the directory `/home/george/subversion/repository` as the repository. `svnserve` is easy to use, but it is not capable of handling large amounts of traffic. If you need to handle a lot of traffic, use Apache and `mod_dav` instead.

■ **Note** While deploying the application, you are working on two machines: the local workstation and the remote server. Whenever you edit or create a file, either locally or on the server, remember to commit the changes to Subversion. Capistrano, which we will demonstrate later in this chapter, will use the latest version found in Subversion when deploying to production.

You could also use SCP, which is distributed with the OpenSSH package, to copy the files from your local machine to the production server:

```
$ scp -r /home/george/projects/emporium \
george@production_machine:/u/apps/emporium/current/
```

Creating Users and Groups

We'll create two different users and groups: one will be the owner of the LightTPD process, and the other will be the owner of the FastCGI and `spawner` processes. (The `spawner` is a separate process running in the background that makes sure that the specified amount of FastCGI processes is running at all times.) Log in to the remote machine and execute the following commands:

```
$ sudo addgroup rails
$ sudo useradd -g rails -d /home/lighttpd rails
```

This creates the `rails` user and group, and sets the `home` directory to `/home/rails`.

Next, create the `home` directory and set the access rights with the following commands:

```
$ sudo mkdir /home/rails
$ sudo chown rails /home/rails
$ sudo chgrp rails /home/rails
```

You should also add the `rails` user to the list of `sudoers` by executing the `visudo` command:

```
$ sudo visudo
```

This allows the user to gain administrator rights through the `sudo` command. The `visudo` command opens the file `/etc/visudoers` in an editor. Add the following to the end of that file:

```
%rails ALL=(ALL) ALL
```

We also need to create the user and group that will own the LightTPD processes. This is done by executing the following commands on the production machine:

```
$ sudo addgroup lighttpd
$ sudo useradd -g lighttpd -d /home/lighttpd lighttpd
```

The first command creates the `lighttpd` group. The second creates a user named `lighttpd` that belongs to the `lighttpd` group. We also specify the `home` directory for the new user account, which doesn't exist yet. Create this directory and give the appropriate rights to it by executing the following commands:

```
$ sudo mkdir /home/lighttpd
$ sudo chown lighttpd /home/lighttpd
$ sudo chgrp lighttpd /home/lighttpd
```

Note that the user that LightTPD runs under is specified in the configuration file with the `server.username` and `server.groupname` options.

You should also set a sensible password for both the `lighttpd` and `rails` users using the `passwd` command, as shown here:

```
$ sudo passwd rails
```

```
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Starting LightTPD

Now that we have the project on the production machine, we can start LightTPD by using the `lighttpd` command and specifying the location of the configuration file:

```
$ sudo lighttpd -f /u/apps/emporium/current/config/lighttpd-production.conf
```

We start LightTPD using the `sudo` command because the server needs to bind to the restricted port 80. You can verify that LightTPD is running by executing the following command:

```
$ ps -ef|grep lighttpd
```

```
lighttpd  2325     1  0 13:53 ?        00:00:00 lighttpd -f /u/apps/emporium/…
```

Note that the process is running under the `lighttpd` user we just created, even though we started it using the `sudo` command and another user. This is because we set the `server.username` and `server.groupname` parameters to `lighttpd` in the `lighttpd` configuration file.

Starting FastCGI Processes

Before starting the FastCGI processes, we need to change the group ownership of the application directory to `rails` with the following command:

```
$ sudo chgrp -R rails /u/apps/emporium/current/
$ sudo chown -R rails /u/apps/emporium/current/
```

Next, start the FastCGI processes (and the application) by executing the following commands:

```
$ su - rails
```

```
Password:
```

```
$ /u/apps/emporium/current/script/process/spawner -p 7000 -i 4
```

```
Checking if something is already running on port 7000...NO
 Starting FCGI on port: 7000
  spawn-fcgi.c.190: child spawned successfully: PID: 4289
Checking if something is already running on port 7001...NO
 Starting FCGI on port: 7001
  spawn-fcgi.c.190: child spawned successfully: PID: 4291
Checking if something is already running on port 7002...NO
 Starting FCGI on port: 7002
  spawn-fcgi.c.190: child spawned successfully: PID: 4293
Checking if something is already running on port 7003...NO
 Starting FCGI on port: 7003
  spawn-fcgi.c.190: child spawned successfully: PID: 4295
```

The spawner script automatically starts the number of FastCGI processes specified with the i parameter. As you can see from the example, the spawner script starts four processes listening on ports 7000, 7001, 7002, and 7004.

Capistrano expects to find a "spinner" script that it can execute to start (spawn) your application, so save the following in scripts/spin:

```
/u/apps/emporium/current/script/process/spawner -p 7000 -i 4 --repeat=60
```

Notice that we added the repeat option, which keeps the spawner process running. The spawner checks (every 60 seconds) that all FastCGI processes are running, and starts them again if they crash.

■ **Note** Remember to make the scripts executable by running chmod ug+x script/process/*, before checking in the files to Subversion. Subversion will preserve the access rights after adding the files to source control. This means that the scripts can be executed when the code is checked out by Capistrano on the production server.

Next, run the spin script:

```
$ scripts/spin
```

Verify that there are one lighttpd process, a spawner process, and four FastCGI ruby processes up and running with the following command:

```
$ ps -ef | grep 'ruby\|lighttpd'
```

```
lighttpd  2325     1  0 13:53 ?        00:00:00 lighttpd -f ➥
/u/apps/emporium/current/config/lighttpd-production.conf
rails     4557     1  0 15:03 ?        00:00:00 ruby ➥
 /u/apps/emporium/current/script/process/spawner ➥
-p 7000 -i 4 --repeat=10
rails     4559     1  0 15:03 ?        00:00:01 /usr/bin/ruby1.8 ➥
/u/apps/emporium/current/public/dispatch.fcgi
rails     4561     1  0 15:03 ?        00:00:01 /usr/bin/ruby1.8 ➥
/u/apps/emporium/current/public/dispatch.fcgi
rails     4563     1  0 15:03 ?        00:00:01 /usr/bin/ruby1.8 ➥
 /u/apps/emporium/current/public/dispatch.fcgi
rails     4565     1  0 15:03 ?        00:00:01 /usr/bin/ruby1.8 ➥
/u/apps/emporium/current/public/dispatch.fcgi
```

The first column in the output tells you the owner of the process. You can now try to access your production environment by opening one of Emporium's pages, such as http://production_server/catalog, in your browser. You should see the Emporium catalog load. If not, examine the Rails and LightTPD error logs to see what went wrong.

Phew! Remembering all of the commands required to deploy a Rails application is tough. This is where Capistrano comes in handy. In the next section, we'll automate the deployment with Capistrano.

Automating Deployment

Capistrano is a tool aimed at automating deployment tasks, but it can also do a lot more. Capistrano allows you to execute almost any command simultaneously on multiple remote servers. For example, it can restart a web server, create a database, deploy your application, and start the FastCGI dispatchers at the same time on one or more remote servers. Instead of remembering a long series of commands, you can deploy a new version of your application with just one command: `rake deploy`. This makes it easy to deploy new versions or simple bug fixes. Capistrano does everything in transactions and supports rollback, so that if something fails on one server, the changes on all servers are rolled back. Capistrano also lets you roll back to the previous version of an application after deployment (by running `cap rollback`).

Capistrano requires that you have an SSH server running on your remote server. (Setting up an SSH server was discussed earlier in this chapter.) Capistrano first logs in to the remote server using SSH, and then issues shell commands, which are part of a user-defined deployment script, over the secure connection.

Capistrano uses Portable Operating System Interface (POSIX) shell commands, so the operating system on the remote server must be POSIX-based. Linux, Mac OS X, and Unix systems fulfill this requirement, but Windows doesn't.

Installing Capistrano

You can install Capistrano with the RubyGems package system. Simply execute `gem install capistrano` on the command line:

```
$ sudo gem install capistrano --include-dependencies
```

Note that Capistrano will ask you for the password when it establishes the connection to the remote server. By default, Capistrano shows the password in clear text as you type, which is not a good idea if Kevin Mitnick is standing behind you. To hide the password, install the `termios` RubyGem package with the following command:

```
$ sudo gem install termios
```

With Capistrano installed, you're ready to prepare for deployment. The first step is to create the script, or deployment recipe.

Creating the Capistrano Deployment Recipe

Capistrano scripts are called *deployment recipes* and are written in a custom Ruby domain-specific language. Although Capistrano allows you to use the full power of the Ruby language, you probably won't need it, as Capistrano comes with a set of configurable built-in tasks that can be used for the most common requirements. Let's take a quick look at the components of a deployment recipe before we create the Emporium deployment recipe.

Understanding Deployment Recipe Components

The deployment recipe consists of tasks, roles, and variables, which can be customized for your production environment.

Roles

Capistrano allows you to assign roles to servers. For example, all servers running MySQL are assigned the `db` role, servers running LightTPD are assigned the `web` role, and the servers running the FastCGI processes are assigned the `app` role.

Roles allow you to target tasks to be run only on servers having a specific role. Roles can be defined in the deployment recipe, as shown here:

```
role :web, "www.emporium.com"
role :app, "app1.emporium.com", "app2.emporium.com"
role :db, "db.emporium.com"
```

Variables

The lead developer of Capistrano and Rails core team member, Jamis Buck, embraced the familiar "convention over configuration" rule when writing Capistrano. For example, Subversion is the default version control system, but you can change it to any of the supported ones by modifying a variable in the deployment recipe. You can also declare your own variables and use them in custom tasks. For example, the following lines set the application name to Emporium and the Subversion repository URL to `svn://localhost/emporium/trunk`.

```
set :application, "Emporium"
set :repository, svn://localhost/emporium/trunk
```

Capistrano comes with a set of predefined variables. The following are three of the more commonly used variables:

- `application`: The name of your application, such as Emporium.
- `repository`: The location of your application's source managed by a version control system, such as a Subversion URL: `svn://localhost/emporium/trunk`.
- `user`: The name to use when logging in to the remote server. Note that Capistrano uses the same name when logging in to all servers. This means that the user must exist on all servers.

Tasks

Capistrano has a set of built-in tasks that can be used to perform work on the remote server. The `deploy` task, for example, installs and deploys a new version of your application on the remote machine. The `deploy` task itself calls the `restart` task to restart FastCGI processes and other tasks to complete the deployment.

You can also add your own custom tasks to the deployment recipe. Tasks are written in Ruby. For example, you could create a task that runs the `mysqldump` command on the remote machine, as shown in this example:

```
task :backup_production_database do
  run "mysqldump -uemporium -phacked emporium_production >> ➥
/var/emporium/production_backup.sql"
end
```

You can get a list of all available tasks by executing `rake remote:show_tasks`.

A task is run on all servers by default. Specify roles to run a task on a specific server or group of servers.

Generating the Deployment Recipe

The first thing you need to do is create the deployment recipe, by applying Capistrano to your application. To do this, execute the `cap --apply-to` command on your local machine:

```
$ cap --apply-to /home/george/project/emporium Emporium
```

```
      exists  config
      create  config/deploy.rb
      exists  lib/tasks
      create  lib/tasks/capistrano.rake
```

The command creates two files: `deploy.rb` is your deployment recipe, and `capistrano.rake` is an extension to `rake` that allows you to run all the tasks in your deployment recipe with `rake`. If you run `rake -T` now, you can see that Capistrano added a lot of new tasks, some of which are shown here:

```
$ rake -T
```

```
rake remote:cleanup
rake remote:cold_deploy
rake remote:deploy
rake remote:deploy_with_migrations
rake remote:diff_from_last_deploy
rake remote:disable_web
rake remote:enable_web
```

Modifying the Deployment Recipe

Now that you have the deployment recipe, you can start modifying it to fit your environment. First, set the required variables. Open `config/deploy.rb` in your editor and change the required variables section as shown here:

```
# ==========================================================================
# REQUIRED VARIABLES
# ==========================================================================
set :application, "emporium"
set :repository, "svn://localhost/emporium/trunk"
```

The `application` variable is used when creating the directory structure. The `repository` variable should be set to point to your Subversion repository.

Next, define roles. Recall that we have the web, application, and database servers deployed on the same machine, so define the three different roles shown in this example (remember to change the IP address to fit your environment):

```
# ==========================================================================
# ROLES
# ==========================================================================
role :web, "192.168.0.1"
role :app, "192.168.0.1"
role :db,  "192.168.0.1", :primary => true
```

We can now target a command to be run on the web, application, or database server. We can also add servers to the environment, and the deployment of the application would still remain the same; only the configuration would change. For example, to add two more application servers to the environment, we might change the configuration as follows:

```
role :app, "192.168.0.1", "192.168.0.11", "192.168.0.12"
```

Also note that we have set the one and only database server to be the primary server, so that we can run migrations on it.

Our particular production environment requires some modifications to the Capistrano default settings, so change the optional settings section as shown here:

```
# ==========================================================================
# OPTIONAL VARIABLES
# ==========================================================================
set :user, "rails"            # defaults to the currently logged in user
set :spinner_user, 'rails'
set :svn_username, "svn"
set :svn_password, "hacked"
```

By default, Capistrano will take the username that you used to log in to the machine you are running Capistrano from and use it to log in to the remote system. So, for example, if you logged in as `george` to your workstation, Capistrano would use that username. You usually want to have one dedicated user for deploying applications to production, and this is most likely not the same as your own username. By specifying the `user` variable, we are telling Capistrano to use the `rails` user, which we created earlier, to log in to the remote system.

Recall that the FastCGI processes will be started by the spinner script (`script/spin`) that we created earlier. By default, Capistrano is configured to start the spinner script with the `app` username. But we also want the spinner process to run as the `rails` user, so we have added `spinner_user` to the optional variables section.

`svn_username` and `svn_password` are used by Capistrano to log in to the Subversion repository when checking out the code. Change them as appropriate.

Running the Setup Task

To prepare the production server for deployment, we'll use the `setup` task, which creates the required directory structure. To simulate a deployment to a clean environment, first delete the `/u` directory we created earlier:

```
$ sudo rm -rf /u
```

Then kill any Ruby or LightTPD processes that might be running with the following commands:

```
$ sudo killall -9 /usr/bin/ruby1.8
$ sudo killall -9 lighttpd
```

By default, Capistrano deploys your application to the `/u` directory. This directory can be created only by the root user, which means the deployment will fail if you run it now. To fix this, we'll use a Capistrano `before` filter that creates the directory and then changes the access rights. Add the following task to the end of the deployment recipe (`config/deploy.rb`):

```
task :before_setup do
  sudo "mkdir -m 770 /u"
  sudo "chgrp rails /u"
end
```

Note that we use the `sudo` command to create the directory. Earlier, we also added the `rails` user to the `sudoers` list.

Now run the `setup` task in the root of the application directory:

```
$ cap -a setup
```

```
    loading configuration /usr/lib/ruby/gems/1.8/gems/capistrano- ➥
1.1.0/lib/capistrano/recipes/standard.rb
    loading configuration ./config/deploy.rb
  * executing task before_setup
  * executing "sudo mkdir -m 770 /u"
    servers: ["192.168.0.1"]
Password:
    [192.168.0.1] executing command
    command finished
  * executing "sudo chgrp rails /u"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
    command finished
  * executing task setup
  * executing "mkdir -p -m 775 /u/apps/emporium/releases ➥
/u/apps/emporium/shared/system &&\n ➥
mkdir -p -m 777 /u/apps/emporium/shared/log"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
    command finished
```

By inspecting the output of the command, you can see that it executes the `before_setup` task we added. If you log in to the remote machine, you will notice that Capistrano created the following directory structure:

```
$ tree /u
```

```
/u
`-- apps
    `-- emporium
        |-- releases
        `-- shared
            |-- log
            `-- system

6 directories, 0 files
```

Note that you need to run the `setup` task only once.

Deploying to Production

Now that we have run the `setup` task, we can continue and start the Emporium application on the production server. We could check out the source from Subversion and execute the `spawner` script manually, but instead, we'll introduce you to the `cold_deploy` Capistrano task.

The cold_deploy task does exactly what we need: first it executes the Capistrano deploy task and then the spawner task. Recall that the deploy task checks out the source code on the remote machine and the spinner task starts the FastCGI processes. Execute rake as shown here:

```
$ rake remote:cold_deploy
```

```
 (in /home/george/projects/emporium)
    loading configuration /usr/lib/ruby/gems/1.8/gems/capistrano- ➥
1.1.0/lib/capistrano/recipes/standard.rb
    loading configuration ./config/deploy.rb
  * executing task cold_deploy
  * executing task deploy
 ** transaction: start
  * executing task update_code
  * querying latest revision...
  * executing "if [[ ! -d /u/apps/emporium/releases/20060813212006 ]]; ➥
then\n              svn co --username svn -q -r7
svn://localhost/emporium/trunk /u/apps/emporium/releases/20060813212006 ➥
&&\n              (test -e /u/apps/emporium/revi
sions.log || touch /u/apps/emporium/revisions.log && chmod 666 ➥
/u/apps/emporium/revisions.log) && echo `date +\"%Y-%m-%d
 %H:%M:%S\"` $USER 7 20060813212006 >> /u/apps/emporium/revisions.log;\n ➥
            fi"
    servers: ["192.168.0.1"]
Password:
    [192.168.0.1] executing command
    command finished
  * executing "rm -rf /u/apps/emporium/releases/20060813212006/log ➥
 /u/apps/emporium/releases/20060813212006/public/syste
m &&\n    ln -nfs /u/apps/emporium/shared/log ➥
/u/apps/emporium/releases/20060813212006/log &&\n ➥
ln -nfs /u/apps/empor
ium/shared/system /u/apps/emporium/releases/20060813212006/public/system"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
    command finished
  * executing task symlink
  * executing "ls -x1 /u/apps/emporium/releases"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
    command finished
  * executing "ln -nfs /u/apps/emporium/releases/20060813212006 ➥
/u/apps/emporium/current"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
    command finished
```

```
 ** transaction: commit
  * executing task restart
  * executing "sudo /u/apps/emporium/current/script/process/reaper"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
 ** [out :: 192.168.0.1] Couldn't find any process matching: ➥
/u/apps/emporium/current/public/dispatch.fcgi
    command finished
  * executing task spinner
  * executing "sudo -u rails /u/apps/emporium/current/script/spin"
    servers: ["192.168.0.1"]
    [192.168.0.1] executing command
    command finished
```

Capistrano will also restart existing FastCGI processes with the `reaper` script, as you can see from the output of the `rake` command.

Next, log in to the remote machine and use the `tree` command to display the directory structure created by Capistrano.

```
$ tree /u
```

```
/u
`-- apps
    `-- emporium
        |-- current -> /u/apps/emporium/releases/20060516214952
        |-- releases
        |   `-- 20060516214952
        |-- revisions.log
        `-- shared
            |-- log
            `-- system
```

The `apps` directory contains a separate directory for all applications that have been deployed. Now there's only one directory for Emporium, which contains a subdirectory named `releases`.

The `releases` directory is where your application is deployed into a directory named after the time and date the build was created. The `releases` directory is not referred to directly by scripts; instead, they refer to the symbolic link `current`. The `current` link is updated by the `deploy` task and points to the latest version of your application.

The next time you deploy your application, you can run `rake deploy` instead of `rake cold_deploy`.

■ **Note** You should configure the Ferret search engine (introduced in Chapter 4) to store the index outside your application directory (for example, `/u/apps/emporium/shared`). This is because Capistrano deploys your application to a different directory each time you perform a deployment. If the index were in the application directory, Ferret wouldn't be able to find it, and would create a new, empty one. One option is to put the indices in `shared/index`, and create an `after_deploy` hook that creates a `symlink` from `current/index` to `shared/index`.

Capistrano also has a built-in task that you can use for running database migrations during deployment on the remote machine. This means that you don't need to create the database schema yourself, as you do when you deploy your application manually. Use the following command to run migrations along with the deployment:

```
$ rake remote:deploy_with_migrations
```

This checks out the latest version of the source on the remote machine. After the checkout has completed, `rake` runs the migrations, which create the Emporium database.

Starting LightTPD

The last step we need to perform is to start up LightTPD, which acts as a reverse proxy for the FastCGI processes. Again, we'll create a new task in our deployment recipe to save us the trouble of having to manually log in to the remote server(s) and execute the command each time we want to start the web server.

Add the following task to the deployment recipe (`config/deploy.rb`):

```
task :start_lighttpd, :roles => 'web' do
  sudo "lighttpd -f /u/apps/emporium/current/config/lighttpd-production.conf"
end
```

Notice that we have told Capistrano to run the task only on servers having the `web` role.

Next, run the task by executing the following command:

```
$ cap -a start_lighttpd
```

```
    loading configuration /usr/lib/ruby/gems/1.8/gems/capistrano- ➥
1.1.0/lib/capistrano/recipes/standard.rb
    loading configuration ./config/deploy.rb
  * executing task start_lighttpd
  * executing "sudo lighttpd -f ➥
/u/apps/emporium/current/config/lighttpd-production.conf"
    servers: ["192.168.0.1"]
Password:
    [192.168.0.1] executing command
    command finished
```

You should see the script complete successfully.

■ **Tip** You should need to start LightTPD web server only once. Rebooting the machine will, of course, kill your processes, so remember to create a start script that runs at reboot and that starts LightTPD and the `spawner` process.

Open Emporium in your browser and do a quick test. You shouldn't see any errors, which means that you have completed the deployment.

Before we wrap up this chapter, we should tell you that FastCGI processes are known to start acting crazy once in a while. The only option, usually, is to restart the processes. This is one of the reasons why you should install a system monitoring tool like Nagios (`http://nagios.org/`) or monit (`www.tildeslash.com/monit/`). These tools help you notice when things go bad—not only with FastCGI, but also with other processes and protocols.

Summary

In this chapter, you learned how to set up a real-world production environment. We showed you how to install LightTPD and FastCGI by compiling from source. We also explained how to configure LightTPD for use in a production environment. Then we showed you how to deploy an application manually. Finally, you saw how to automate the deployment process with Capistrano, which makes your life as a developer easier and drastically lowers the barrier for deploying new features into production (at least for procrastinators).

In the next chapter, we'll show you how to tune an application's performance.

CHAPTER 13

Performance Optimization

We now have a working application, and people are already rambling in to shop at the Emporium online store. While George is very happy that the money is flowing in, he has noticed that the site is behaving less responsively lately.

Here, we will look at techniques for optimizing the application. This chapter is not about specific functionality in the application, so we won't tell it in the form of user stories. Neither do we use TDD here, since we're not really developing anything.

Performance and Scaling

Recently, the terms *performance* and *scaling* have been used interchangeably within the context of web development. Actually, they don't mean the same thing.

Performance means how many concurrent users can use a web application and still consider it working fast enough. The "fast enough" part depends on the application and its use.

Scaling, on the other hand, is a totally different beast. To paraphrase the creator of Mongrel, Zed Shaw (`http://www.oreillynet.com/ruby/blog/2006/05/post.html`), scaling should be more analogous to "resource-expandable," meaning that you can start with a moderate hardware and software stack, and easily expand it so that the application is snappy, even if it gets Slashdotted.

An often-heard argument against Rails is that it doesn't scale. People stating this often assume that scaling and performance are the same thing, and really mean that Rails is slow. Although Ruby as an interpreted, dynamic language is not among the fastest programming languages, that doesn't mean Rails as a web framework is slow. The fact that 37signals ran Basecamp with up to tens of thousands of customers on a single box without any caching whatsoever should be enough to prove that Rails is fast enough for most uses.

There is nothing in Rails that would make it inherently hard to scale (scale as in being easily "resource-expandable")—quite the contrary. Rails uses *shared nothing architecture* similar to that used by many very high-traffic websites like Google and Livejournal. Shared nothing is a distributed architecture consisting of independent nodes without a single point of contention. In Rails, this means that you can scale very easily by adding new application servers. If the pages are not rendered fast enough by one server, add another server next to it. Load balance requests between the two, and you have roughly doubled the performance of the application (depending on the database performance).

Measuring Performance

An old saying goes, "You can't manage what you don't measure." You need to know what to optimize in order to optimize right things.

We have a plethora of options for measuring the performance of a Rails application. Here, we will look at what the log files can tell us, and then try out the Rails Analyzer tool set.

Checking the Log File

To use the simplest way to "profile" your application, you don't need any extra tools. Just take a look at the `development.log` file in the log directory of your application. Here is sample output of loading the show page for a book:

```
Processing CatalogController#show (for 127.0.0.1 at 2006-09-27 23:45:30) [GET]
  Session ID: 83bd38942bba92536d89eabe192877a4
  Parameters: {"action"=>"show", "id"=>"17",
"controller"=>"catalog"}
   [4;36;1mCart Load (0.000750) [0m    [0;1mSELECT *
FROM carts WHERE (carts.id = 2) LIMIT 1 [0m
   [4;35;1mBook Load (0.007462) [0m    [0mSELECT *
FROM books WHERE (books.id = '17') LIMIT 1 [0m
   [4;36;1mBook Columns (0.000859) [0m    [0;1mSHOW
FIELDS FROM books [0m
Rendering  within layouts/application
Rendering catalog/show
   [4;35;1mJoin Table Columns (0.084852) [0m    [0mSHOW
FIELDS FROM authors_books [0m
   [4;36;1mAuthor Load (0.001258) [0m    [0;1mSELECT *
FROM authors INNER JOIN authors_books ON authors.id =
authors_books.author_id WHERE (authors_books.book_id = 17 )  [0m
   [4;35;1mAuthor Columns (0.000625) [0m    [0mSHOW FIELDS
FROM authors [0m
   [4;36;1mCart Columns (0.000682) [0m    [0;1mSHOW FIELDS
FROM carts [0m
   [4;35;1mCartItem Load (0.001169) [0m    [0mSELECT * FROM
cart_items WHERE (cart_items.cart_id = 2)  [0m
Rendered cart/_cart (0.08493)
Completed in 1.78054 (0 reqs/sec) | Rendering: 0.17748 (9%)
| DB: 0.09766 (5%) | 200 OK [http://localhost/catalog/show/17]
```

From the output, you can see the times it took to run individual SQL queries. The final lines show a summary of the whole action.

Here, we don't see any glaring performance hogs. However, if a single database query took considerably more time than others, it would be a good starting point for optimization.

Note that only the development environment outputs this much detail in the log file.

Using Rails Analyzer

The Robot Co-op, the creators of the popular 43things.com, 43places.com, and 43people.com, has released an impressive set of tools for analyzing the performance of a Rails application. The tool set is called Rails Analyzer and can be found at `http://rails-analyzer.rubyforge.org/`.

Rails Analyzer consists of four independent parts:

- The Production Log Analyzer can be used to analyze Rails log files. It produces a report that tells which actions are the most popular and which take the most time to render. It is an invaluable tool for measuring the performance of those actions that are used most often in the production setting.
- The Action Profiler is used to profile individual actions. Run it as the next step after using the Production Log Analyzer to find the slow actions. With the Action Profiler, you can drill down to the action and see where it is taking its time.
- Rails Analyzer Tools is a collection of tools for monitoring and benchmarking a Rails application. The tools included are `bench` for benchmarking a particular page, `crawler` for crawling a page and requesting all the local linked files on that page, and `rails_stat` for pinging the load status of a live production Rails application.
- SQL Dependency Grapher can be used to visualize the frequency of table dependencies in a Rails application. (We won't cover this part of the tool set in this chapter.)

Let's see how Rails Analyzer can help us. First, log on to the production server and install the needed gems:

```
$ sudo gem install rails_analyzer_tools production_log_analyzer action_profiler
```

Now we can get started by setting up for the Production Log Analyzer and then running it.

Using the Production Log Analyzer

The Production Log Analyzer can't use the standard Rails log format because it needs a log line to be able to be clearly identified with a single action. So, we need to substitute the logger with `SyslogLogger`, a class provided by the Rails Analyzer Tools gem. Note that this works only on Linux and FreeBSD. If you use some other production environment, consult the Rails Analyzer homepage for more information.

Add the following lines to `config/environments/production.rb`:

```
require 'analyzer_tools/syslog_logger'
RAILS_DEFAULT_LOGGER = SyslogLogger.new
```

This tells Rails to use the replacement logger when in production.

Next, add the following lines to `/etc/sysconf.log`:

```
!rails
 *.*                    /var/log/production.log
```

This tells `syslog` to log all Rails-related entries to your own production log file.

Now create the file and restart the `syslog` daemon:

```
sudo touch /var/log/production.log
sudo killall -HUP syslogd
```

Restart the Rails application, and it should be logging to `/var/log/production.log`. Confirm this by tailing the log file (with `tail -f`, for example) and loading a page on the site. You should see something like the following show up at the end of the output:

```
Sep 28 00:46:22 emporium rails[6975]: Rendered cart/_cart (0.09150)
Sep 28 00:46:22 emporium rails[6975]: Completed in 0.53988
(1 reqs/sec) | Rendering: 0.37126 (68%) | DB: 0.05668 (10%) | 200
OK [http://emporium.com/catalog/show/17]
```

Now you know that the logging is working in the way it needs to for the Production Log Analyzer.

Let's wait overnight and see how much traffic George gathers on the site. (If you are eager to get going, write a little script that loads the pages randomly, or go the manual route by browsing around the site a few times.)

Woke up already? Good, not bad for a hacker. Let's get back to work. Run the Production Log Analyzer to see what happened while we were asleep:

```
$ sudo pl_analyze /var/log/production.log
```

```
Request Times Summary:          Count   Avg     Std Dev Min     Max
ALL REQUESTS:                   266     0.115   0.344   0.000   2.066

Unknown:                        197     0.000   0.000   0.000   0.000
CatalogController#index:        28      1.047   0.378   0.550   2.066
CatalogController#show:         21      0.034   0.113   0.005   0.540
AboutController#index:          20      0.023   0.016   0.013   0.077

Slowest Request Times:
        CatalogController#index took 2.066s
        CatalogController#index took 1.750s
        CatalogController#index took 1.654s
        CatalogController#index took 1.504s
        CatalogController#index took 1.476s
        CatalogController#index took 1.392s
        CatalogController#index took 1.388s
        CatalogController#index took 1.237s
        CatalogController#index took 1.222s
        CatalogController#index took 1.219s

------------------------------------------------------------------------
--
```

```
DB Times Summary:               Count   Avg     Std Dev Min     Max
ALL REQUESTS:                   266     0.006   0.019   0.000   0.165

Unknown:                        197     0.000   0.000   0.000   0.000
CatalogController#index:        28      0.052   0.032   0.024   0.165
CatalogController#show:         21      0.003   0.012   0.000   0.057
AboutController#index:          20      0.000   0.000   0.000   0.001

Slowest Total DB Times:
        CatalogController#index took 0.165s
        CatalogController#index took 0.156s
        CatalogController#index took 0.077s
        CatalogController#index took 0.063s
        CatalogController#index took 0.062s
        CatalogController#index took 0.058s
        CatalogController#index took 0.057s
        CatalogController#show took 0.057s
        CatalogController#index took 0.054s
        CatalogController#index took 0.054s

------------------------------------------------------------------------
--

Render Times Summary:           Count   Avg     Std Dev Min     Max
ALL REQUESTS:                   266     0.099   0.306   0.000   1.934

Unknown:                        197     0.000   0.000   0.000   0.000
CatalogController#index:        28      0.915   0.377   0.429   1.934
CatalogController#show:         21      0.018   0.079   0.000   0.371
AboutController#index:          20      0.020   0.010   0.013   0.055

Slowest Total Render Times:
        CatalogController#index took 1.934s
        CatalogController#index took 1.554s
        CatalogController#index took 1.524s
        CatalogController#index took 1.403s
        CatalogController#index took 1.396s
        CatalogController#index took 1.266s
        CatalogController#index took 1.208s
        CatalogController#index took 1.122s
        CatalogController#index took 1.102s
        CatalogController#index took 1.100s
```

The first listing in the output is a summary of the complete request times. After that are similar summaries for the times needed in the database and for the rendering. A summary lists the different requested actions, the count of times they were requested, and the statistics of their performance (including average, minimum, and maximum times and the standard deviation). From this listing, it is pretty easy to see which actions tend to be the slowest. If a slow action is also a very popular action, it is a good candidate for being optimized in one way or another.

Following the statistical analysis is a list of the slowest individual action loads in each category. An astute reader can see that the slowest request time should match the `Max` field value in the `ALL REQUESTS` row in the statistical analysis.

From the analysis, you can see that the three actions that had been requested are pretty much equally popular. However, the index page in `CatalogController` is clearly the slowest, and its request times are more than one second on average.

Next, take a look at what happens in the action by using the `action_grep` command included in the Production Log Analyzer package:

```
$ sudo action_grep CatalogController#index /var/log/production.log
```

```
Sep 28 12:17:50 emporium rails[10500]: Processing CatalogController#index
 (for 81.193.72.157 at 2006-09-28 12:17:50) [GET]
Sep 28 12:17:50 emporium rails[10500]: Session ID:
ba0a2e9b205ed0da9390dc08ea00d114
Sep 28 12:17:50 emporium rails[10500]: Parameters: {"action"=>"index",
 "controller"=>"catalog"}
Sep 28 12:17:50 emporium rails[10500]: Globalize::Language Columns
(0.002259)   SHOW FIELDS FROM globalize_languages
Sep 28 12:17:50 emporium rails[10500]: Globalize::Language Load (0.000155)
   SELECT * FROM globalize_languages WHERE (globalize_languages.`iso_639_1`
 = 'en' ) LIMIT 1
Sep 28 12:17:50 emporium rails[10500]: Cart Load (0.000108)   SELECT * FROM
 carts WHERE (carts.id = 107) LIMIT 1
Sep 28 12:17:50 emporium rails[10500]: Book Count (0.000156)   SELECT
COUNT(DISTINCT books.id) FROM books LEFT OUTER JOIN authors_books
ON authors_books.book_id = books.id LEFT OUTER JOIN authors
ON authors.id =authors_books.author_id LEFT OUTER JOIN publishers
ON publishers.id = books.publisher_id
Sep 28 12:17:50 emporium rails[10500]: Globalize::Language Load (0.000650)
SELECT * FROM globalize_languages WHERE (globalize_languages.`rfc_3066` =
'en-US' ) LIMIT 1
Sep 28 12:17:50 emporium rails[10500]: Globalize::Language Load (0.000140)
SELECT * FROM globalize_languages WHERE (globalize_languages.`iso_639_1` =
'en' ) LIMIT 1
Sep 28 12:17:50 emporium rails[10500]: Book Columns (0.002050)   SHOW FIELDS
FROM books
Sep 28 12:17:50 emporium rails[10500]: Author Columns (0.001774)   SHOW FIELDS
FROM authors
```

```
Sep 28 12:17:50 emporium rails[10500]: Publisher Columns (0.001681)   SHOW
FIELDS FROM publishers
Sep 28 12:17:50 emporium rails[10500]: Book Load IDs For Limited Eager
Loading (0.000133)   SELECT id FROM books ORDER BY books.id desc LIMIT 0, 10
```

The command gives a lot of output about what has been happening in the `index` actions in `CatalogController`. You can see that most of the database traffic is related to the Globalize plugin. However, if you take a closer look at the Production Log Analyzer output, you will see that most of the time is not spent in the database, but in rendering the page (0.052 second vs. 0.915 second). We should find out what is taking so much time in rendering the page. And that's where Action Profiler comes in.

Running the Action Profiler

The Action Profiler can profile a single Rails action call. It works with any of the three profiler tools for *nix machines: the built-in Ruby profiler, ruby-prof (`http://ruby-prof.rubyforge.org/`), or ZenProfiler (`http://rubyforge.org/projects/zenhacks/`). If you want to use ruby-prof or ZenProfiler, you need to install it first.

Here's an example of running the Action Profiler with the built-in profiler against the `show` action in `CatalogController`:

```
$ action_profiler -P Profiler CatalogController#show
```

```
Warmup...
Profiling...
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 26.98     1.02      1.02      204     5.00     6.37  String#scan
 12.17     1.48      0.46      408     1.13     1.94  Pathname#initialize
 10.32     1.87      0.39      204     1.91    11.62  Pathname#cleanpath_aggressive
  5.82     2.09      0.22      424     0.52     2.52  Class#new
  5.56     2.30      0.21     2587     0.08     0.08  String#==
  4.50     2.47      0.17      306     0.56    10.52  String#gsub
  3.70     2.61      0.14      614     0.23     0.44  Kernel.dup
  3.44     2.74      0.13      613     0.21     0.21  String#initialize_copy
... a lot of output ...
```

You can see from the output that about half of the time is spent scanning strings and creating path names. However, since the built-in profiler presents everything in a flat output, it is hard to know in which part of the application code the time is actually spent. If you want output with call graphs to see where different methods are called from, consider installing ruby-prof.

■ **Note** There are currently some compatibility issues between the Action Profiler and both ZenProfiler and ruby-prof. We hope they will be resolved by the time this book hits the shelves.

RAILSBENCH

Another tool for measuring Rails application performance is Railsbench by Rails performance expert Stefan Kaes. You can download if from `http://railsbench.rubyforge.org/`.

If you're working on Windows, you might find Railsbench more useful than the Action Profiler. By default, it uses the Windows-only Ruby Performance Validator (`www.softwareverify.com/ruby/profiler/index.html`), which is said to be the best Ruby profiler around.

Caching

Now that we have measured the performance of our application and tracked slow actions, it's time to do something about them. One of the most common ways to speed up a website is to use some kind of caching. If an often-viewed page needs a large amount of database queries or otherwise expensive calculations to produce its output, storing the output in a cache can make the site a lot more responsive.

Out of the box, Rails sports three caching levels: page, action, and fragment caching. You can also use the `cached_model` library for caching ActiveRecord objects, when you need to go beyond the usual caching.

Page Caching

Page caching is the fastest of the caching schemes in Rails. With page caching, the cached page is stored as a static HTML file in the document root of the web server and served directly from there on subsequent requests. This means that Rails can be bypassed altogether, and the web server can serve the page in the same way that it serves other static files. This obviously means a huge impact on the performance. Whereas a single Rails process can serve a few dozen non-cached pages per second, a real web server serving static files can easily reach speeds of up to 1000 requests per second.

Should all our actions then be using page caching? Well, no. Page caching stores the output of an action in a file and serves that same file all the time, to all the people. This has many drawbacks. The page is not really dynamic anymore. You can't use authentication, since Rails is bypassed. You also can't have any personalization on the page. Therefore, page caching is not really recommended for anything but the most static pages served by a Rails application.

Page caching is also kind of against the shared nothing architecture, since the cached pages are stored on the file system. You could use a networked file system that all the application servers would use as the document root, but you would still be vulnerable to different application servers trying to write the same file at the same time.

However, we do have one page in the Emporium application that hardly ever changes: the About page. Let's implement page caching for it as an exercise.

It turns out it is extremely easy to use page caching. The only thing we need to do is add a single row to the controller in question, which in this case is `app/controllers/about_controller.rb`:

```
class AboutController < ApplicationController
  caches_page :index

  def index
    @page_title = 'About Emporium'
  end
end
```

The next time you load the About page, it is stored in `app/public/about.html` and served from there ever after.

We can use the `bench` command (part of the Rails Analyzer tool set) to look at the performance of the page without and with caching (repeat the runs several times until the results stabilize). First try it without caching:

```
$ bench -u http://localhost/about -r 100 -c 10
```

```
100.........90.........80.........70.........60.........50
.........40.........30.........20.........10.........
Total time: 48.2463064193726
Average time: 0.482463064193726
```

And then run `bench` with caching (remember to restart the application to make the code changes live on a production server):

```
$ bench -u http://localhost:3000/about -r 100 -c 10
```

```
100.........90.........80.........70.........60.........50
.........40.........30.........20.........10.........
Total time: 5.81317806243896
Average time: 0.0581317806243896
```

■ **Note** By default caching is turned on only in the production environment. If you want to test it in the development environment, change the `config.action_controller.perform_caching` parameter to `true` in `config/environments/development.rb`.

The difference is noticeable, although not nearly as big as it is in reality, since we're running the `bench` command on the same machine and it is consuming part of the processing power itself.

But what if you want to change the page and expire the cache some time? That's easy, too. Just call `expire_page :action => "index"` in the action where you change the page.

Action Caching

Action caching is the second level of caching in Rails. With action caching, the whole page output is still cached, but this time, the request goes through ActionController, and thus the filters are run before the rendering. The most important consequence of this is that you can use action caching even for pages that need authentication.

Action caching is turned on in the same way as page caching. Add `caches_action :index` to the controller, and call `expire_action :index` to expire the action. You won't get the same raw speed as with page caching, but you will get more flexibility with filtering the requests. The speed would still be plenty fast.

Action caching shares many, but not all, of the problems with page caching. There is still no way to make the page contents dynamic, and complete personalization isn't possible (although action caching can use the user ID as a key in the cached page).

If the results of page caching are stored in the file system, where do the action caches go? The answer is that it depends. Action caching uses internally the third built-in caching scheme in Rails, fragment caching.

Fragment Caching

Fragment caching is the most granular of the standard caching mechanisms in Rails. With it, you can cache parts of a page. For example, you could cache the contents of a shopping cart like this (in `app/views/layouts/application.rhtml`):

```
<% if @cart %>
  <% cache(:controller => "cart", :action => "show",
           :id => @cart) do %>
    <div id="shopping_cart">
    <%= render :partial => "cart/cart" %>
    </div>
  <% end %>
<% end %>
```

This would cause the contents of the cache block to be cached, and we could avoid a perhaps expensive database trip on every request a particular user makes.

The `cache` method takes a hash as its parameter and uses `url_for` to build a URL to be used as a key to the cached item. Needless to say, this should be unique. Note that it doesn't have to be a real, existing URL. In our case, for example, there is no action called `show` in `CartController`. However, the cache key of a stored cart would be something like `emporium.com/cart/show/179`.

Cached fragments are expired with the `expire_fragment` method, which takes a hash as its argument, similar to the argument for the `cache` method. In our case, we need to expire the fragment whenever the shopping cart is changed—when we add or remove books to the cart, clear the cart, or check out.

Let's start with `CheckoutController` (in `app/controllers/checkout_controller.rb`):

```
def place_order
    @page_title = "Checkout"
    @order = Order.new(params[:order])
    @order.customer_ip = request.remote_ip
    populate_order

    if @order.save
      if @order.process
        flash[:notice] = 'Your order has been submitted,
                          and will be processed immediately.'
        session[:order_id] = @order.id
        # Empty the cart
        @cart.cart_items.destroy_all
        expire_fragment(:controller => "cart",
                        :action => "show",
                        :id => cart)
        redirect_to :action => 'thank_you'
      else
        flash[:notice] = "Error while placing
                          order.'#{@order.error_message}'"
        render :action => 'index'
      end
    else
      render :action => 'index'
    end
  end
```

Now whenever an order is processed and the shopping cart is cleared, the cached cart fragment is expired.

For `CartController`, we use a different approach—a *cache sweeper*. A cache sweeper is a special kind of an observer. It observes the lifeline of an object and can sweep cached stuff when specific changes (such as create, update, or destroy) are made to the object in question. Create a file called `cart_sweeper.rb` in `app/models` and add the following code to it:

```
class CartSweeper < ActionController::Caching::Sweeper
  observe Cart, CartItem

  def after_save(record)
    cart = record.is_a?(Cart) ? record : record.cart
    expire_fragment(:controller => "cart",
                    :action => "show",
                    :id => @cart)
  end
end
```

You can see that the sweeper looks just about the same as a normal observer. In this case, we observe both the `Cart` object and the `CartItem` objects that belong to it. When either kind

of object is saved, we find the relevant `Cart` object and expire the fragment that belongs to that cart. To make the sweeper work, we need to call it in `CartController` (`app/controllers/cart_controller.rb`):

```
class CartController < ApplicationController
  cache_sweeper :cart_sweeper
  before_filter :initialize_cart
...
```

That's all. Since we want the sweeper to work on all the actions in `CartController`, we don't have to specify anything else. If we wanted to restrict the sweeper to only certain actions, we could use the `:only` parameter for that.

Fragment Stores

Page caching results are always stored in the file system. Fragment caching has a few more options:

- *File store:* The contents of the cache are stored in the file system, just as in page caching.
- *Memory store:* All the cached fragments are stored in memory. This is the fastest option of all, but it doesn't scale beyond one application server process, since each process keeps its own cache.
- *DRb store:* Cached fragments are kept in the memory of a shared DRb (Distributed Ruby) process. This option scales, but it requires you to build and maintain the process yourself.
- *Memcache store:* Fragments are stored in a memcached process. Is very fast and scales well, since memcached can be accessed through the network.

The fragment store that is being used can be chosen either on the application (in `config/environment.rb`) or environment level (for example, in `config/environments/production.rb`) by setting the `fragment_cache_store` parameter:

```
ActionController::Base.fragment_cache_store = :mem_cache_store, "localhost"
```

■ **Note** Technically, since `fragment_cache_store` is a class variable of the `ActionController::Base` class, you can overwrite it on a controller-specific level as well.

MEMCACHED

According to its website (www.danga.com/memcached/), memcached is "a high-performance, distributed memory object caching system." It was developed by Danga Interactive to help speed the performance of LiveJournal.com, a site serving more than 20 million dynamic page views per day.

Memcached can be used to temporarily store items such as the results of complex database queries and objects that are accessed frequently. Contents of memcached are always stored in RAM, so fetching them is blazingly fast.

Memcached is very lightweight and easy to deploy. It is very memory-hungry and CPU-light, so it is a good companion for an application server, which is often CPU-hungry and lighter on memory. Good advice is to deploy memcached on any server that has free memory, because the load of memcached servers can be distributed.

In a standard Rails setup, you can use memcached to store sessions and cache data. With the libraries introduced later in this chapter, in the "Caching ActiveRecord Objects" section, you can basically cache whatever data you want.

Caching with Memcached

Let's make our application use memcached to store the fragment cache contents. For this, we need to install memcached.

On OS X, you can use Geoffrey Grosenbach's shell script to automate the installation:

```
$ curl -O http://topfunky.net/svn/shovel/memcached/install-memcached.sh
$ sudo sh install-memcached.sh
```

After that, add EVENT_NOKQUEUE=1 to your environment variables. In bash, run the following command:

```
$ echo 'export EVENT_NOKQUEUE=1' >> ~/.bash_profile
```

On other *nix variants, you can use their native package management systems to install memcached or do it by hand. We do the latter, because that way we get the latest version of memcached. Download and compile memcached as follows (check the latest version from danga.com/memcached/download.bml):

```
$ curl -O http://danga.com/memcached/dist/memcached-1.2.0.tar.gz
$ tar zxvf memcached-1.2.0.tar.gz
$ cd memcached-1.2.0
$ ./configure
$ make
$ sudo make install
```

If the `configure` command nags about missing `libevent`, install the `libevent-dev` package (again, with `apt-get` or equivalent, or by hand from `http://www.monkey.org/~provos/libevent/`) and rerun `configure`. Be sure to read the notes in the `README` file in the memcached source distribution if you're on Linux.

On Windows, the support for memcached is a bit flaky. There is no official distribution, but you might want to test if the release at `http://jehiah.com/projects/memcached-win32/` works for you.

After you've installed memcached, you need to install the `ruby-memcache` gem, which is used to interface with memcached:

```
$ sudo gem install ruby-memcache --include-dependencies
```

Now that memcached is installed, you might as well start it. Run it with the `-vv` command to get verbose output of everything that's happening in it.

```
$ memcached -vv
```

```
...
<4 server listening (udp)
```

This will start the memcached server with its default values: port 11211 and 64MB of memory. The server will stay in the foreground, so you can monitor that it's working correctly.

Now you need to make your application use memcached. Add the following line at the end of `config/environments/production.rb`:

```
ActionController::Base.fragment_cache_store = :mem_cache_store, ➥
"localhost"
```

Restart the application server, and the new memcached store should be in use. Try to load a page that includes a shopping cart. You should see memcached output something like the following:

```
<8 new client connection
<8 get emporium.com:3000/cart/show/107
>8 END
<8 set emporium.com:3000/cart/show/107 4 0 311
>8 STORED
```

That's it—our fragment cache is on memcached.

At this point, you would probably want to restart the memcached process, perhaps give it some more memory (128MB should be enough for most Rails applications, but your mileage may vary) with the `-m` option, and make it run in the background using the `-d` option:

```
$ memcached -d -m 128
```

It is also a good idea to monitor the memcached process, just as you watch any other server process. You can use one of the monitoring tools mentioned in the previous chapter for this.

Storing Sessions with Memcached

In Rails, you can use memcached for both caching and storing session data. The reason it would be a good idea to store sessions on memcached, too, is that the data is in a container that can be accessed through a network, in case you later scale the application horizontally by adding new servers. The default session store is file system, which makes scaling possible (using a networked file system), but certainly harder and not as efficient as with memcached.

Making Rails use memcached for storing sessions is another one-liner. Just add the following to `config/environment/production.rb`:

```
config.action_controller.session_store = :mem_cache_store
```

Other options for storing the session data are `:active_record_store`, `:drb_store`, and `:memory_store`, corresponding to the fragment cache store options described previously.

Caching ActiveRecord Objects

While the standard Rails caching mechanisms are useful in many ways, they have their limitations. Page and action caching in particular aren't very useful in applications where the user is shown personalized and highly dynamic content. Fragment caching can be used in many of these cases, but it's kind of cumbersome, and many feel it's done in the wrong place—on the view level.

To address this in a real-world situation, The Robot Co-op has developed and released another library, `cached_model`. `cached_model` creates a new subclass of `ActiveRecord::Base`, which automatically caches the model in memcached.

In our application, an obvious model to cache is `Book`. Another one would be the shopping cart, but we already took care of it with fragment caching.

As usual, installing the `cached_model` gem is simple:

```
$ sudo gem install memcache-client
$ sudo gem install cached_model
```

cached_model uses memcache-client, a replacement client for ruby-memcache. It should be a drop-in replacement and compatible with fragment caching in Rails. However, as of this writing, there were some compatibility issues, so we'll keep the old library around as well.

Now restart the application server and make sure that everything is still working with the new setup.

Next, we need to make our application use the new gem. Add the following to config/environments/production.rb:

```
require 'cached_model'

memcache_options = {
  :c_threshold => 10_000,
  :compression => true,
  :debug => false,
  :namespace => 'emporium_production',
  :readonly => false,
  :urlencode => false
}

CACHE = MemCache.new memcache_options
CACHE.servers = 'localhost:11211'
```

In order to make other environments work with CachedModel, you need to add the preceding lines to all used environments in config/environments. Be sure to give them a different namespace name, though, so that the caches won't clash.

Now all we need to do to make the Book model cached is to change the first line in app/models/book.rb, and make the class inherit from CachedModel instead of ActiveRecord::Base:

```
class Book < CachedModel
  acts_as_taggable
  ...
```

If you now restart the application server and load a book page a couple times in the browser, you should see (if you're still running the memcached server in the foreground) that the book object is stored in memcached on the first request and fetched from there on the subsequent page loads:

```
<10 new client connection
<10 get emporium_production:active_record:Book:15
>10 END
<10 set emporium_production:active_record:Book:15 1 900 193
>10 STORED
<8 get emporium.com:3000/cart/show/107
>8 sending key emporium.com:3000/cart/show/107
>8 END
<10 get emporium_production:active_record:Book:15
>10 sending key emporium_production:active_record:Book:15
>10 END
```

On the second line, memcached checks whether book number 15 exists in the cache. It doesn't, so on the fourth line, it caches the object and confirms this by outputting `STORED`. When the page is requested again, the book (as well as the shopping cart fragment) is found from the cache.

`cached_model` expires the cached model objects automatically every 15 minutes (you can adjust this by setting `CachedModel.ttl`). Since it's tightly integrated with ActiveRecord, expiring objects when they are updated is taken care of automatically.

■ **Tip** For more information about `cached_model` and how to use `memcache-client` directly to cache arbitrary things like collections of objects (`cached_model` doesn't help if you use `find(:all)`, for example), read Geoffrey Grosenbach's excellent tutorial "memcached Basics for Rails" at `http://nubyonrails.com/articles/2006/08/17/memcached-basics-for-rails`.

Common Performance Problems in Rails

The performance of any web application can be severely crippled by small issues in the source code, and a Rails application is no different. Here, we'll look at some of the most common performance problems in Rails applications and how to avoid them.

Rendering Speed

Rendering a page to a user should probably be the fastest and best optimized thing in a web application. Therefore, all really expensive calculations should be made at some other point. One example of this is textilizing content. In Chapter 3, we used the `textilize` method to transform the book blurb stored in Textile format to HTML. However, this formatting can just as well be done whenever the blurb is changed; that is, when the book is saved.

Let's add a field for the blurb in HTML format to the database:

```
$ script/generate migration add_blurb_html_to_books
```

```
      exists  db/migrate
      create  db/migrate/013_add_blurb_html_to_books.rb
```

Next, we open up the migration file we just created and add the new column to it:

```
class AddBlurbHtmlToBooks < ActiveRecord::Migration
  def self.up
    add_column :books, :blurb_html, :text
  end

  def self.down
    remove_column :books, :blurb_html
  end
end
```

Then run the migration:

```
$ rake db:migrate
```

Now we have a column in the database where we can store the textilized blurb. We will do the textilization in a `before_save` filter in the `Book` class. Add the following to `app/models/book.rb`:

```
class Book < CachedModel
  before_save :textilize_blurb

  acts_as_taggable

  [a lot of lines omitted]

  def tagged_with
    tags.collect{|tag| tag.name }.join(", ") if not tags.nil?
  end

  private

    def textilize_blurb
      self.blurb ||= ""
      textilized = RedCloth.new(self.blurb)
      self.blurb_html = textilized.to_html
    end

end
```

Now, whenever a book is saved, its blurb is also transformed to HTML.

We should find all the `textilize` calls in the application and replace them with `blurb_html`. For example, in `app/views/admin/book/show.rhtml`, replace this line:

```
<dd><%= textilize @book.blurb %></dd>
```

with this one:

```
<dd><%= @book.blurb_html %></dd>
```

The result is one less expensive calculation to do in the page-rendering phase.

■ **Caution** Although `textilize` is a cool quick-and-dirty helper, as a rule of thumb, it should *never* be used in a production setting.

Another thing that can slow down rendering is building the URLs in `link_to` and form helpers. If your profiling shows that a `url_for` call is taking a lot of time, you might consider replacing the helper with a handwritten HTML tag. So, for example `<%= link_to "Home", home_url %>` would be replaced by `<a href="/">Home</a>`. However, remember to measure and profile. It serves no purpose to optimize these helpers unless you see that they are guilty of slowing down your application.

Database Access

We already mentioned in Chapter 4 how fetching a list of books and then referencing their authors can lead to $2n+1$ queries, but it's worth a reminder. Use the `:include` parameter in finders if you know you are going to use the associated objects, like authors, on the page:

```
def index
  @book_pages, @books = paginate :books,
                          :per_page => 10,
                          :include => [:authors, :publisher],
                          :order => "books.id desc"
end
```

Another common database performance problem arises when you forget to add needed indices to the database. This is not a Rails-specific problem per se, but it is easy to forget when building an application with small sample data. When the amount of data then grows, you might start wondering why queries that were so snappy when testing are now taking a minute or an hour (no kidding!).

If you track the root of a slow action to a certain database query (or find one by following the slow query log with MySQL, as described in Chapter 10), the next step is to analyze that query. Most database vendors have their own tools for that. In MySQL and PostgreSQL, you can use `EXPLAIN [SELECT query]` for this. As an example, let's run `EXPLAIN` on a query that is run whenever the index page of the catalog controller is loaded (we've truncated the output lines to make them more readable):

```
mysql> EXPLAIN SELECT COUNT(DISTINCT books.id)
    -> FROM books LEFT OUTER JOIN authors_books
    -> ON authors_books.book_id = books.id
    -> LEFT OUTER JOIN authors
    -> ON authors.id = authors_books.author_id
    -> LEFT OUTER JOIN publishers
    -> ON publishers.id = books.publisher_id;
```

```
+----+-------------+--------------+--------+---------------+----
| id | select_type | table        | type   | possible_keys | key
+----+-------------+--------------+--------+---------------+----
|  1 | SIMPLE      | books        | ALL    | NULL          | NULL
|  1 | SIMPLE      | authors_books | ALL    | NULL          | NULL
|  1 | SIMPLE      | authors      | eq_ref | PRIMARY       | PRIMARY
|  1 | SIMPLE      | publishers   | eq_ref | PRIMARY       | PRIMARY
+----+-------------+--------------+--------+---------------+----
4 rows in set (0.00 sec)
```

We're mostly interested in the `type` column of the output. For the two first tables, `books` and `authors_books`, `ALL` means that a full-table scan is done, which should always be at least a bit alarming. If we wanted this query to stay fast when more data is imported in the database, we should make the query use an index for the `authors_books` table. We can do that easily with a migration:

```
$ script/generate migration add_indices
      exists  db/migrate
      create  db/migrate/014_add_indices.rb
```

Then edit the migration file to add needed indices:

```
class AddIndices < ActiveRecord::Migration
  def self.up
    add_index :authors_books, :author_id
    add_index :authors_books, :book_id
  end

  def self.down
    remove_index :authors_books, :author_id
    remove_index :authors_books, :book_id
  end
end
```

We added indices for both foreign key fields. Now let's run the migration:

```
$ rake db:migrate
```

If we now run the `EXPLAIN` query again, we can see that the type of the join is now `ref` instead of `ALL`, meaning that an index is being used to perform the join:

```
mysql> EXPLAIN SELECT COUNT(DISTINCT books.id)
    -> FROM books LEFT OUTER JOIN authors_books
    -> ON authors_books.book_id = books.id
    -> LEFT OUTER JOIN authors
    -> ON authors.id = authors_books.author_id
    -> LEFT OUTER JOIN publishers
    -> ON publishers.id = books.publisher_id;
```

```
+----+-------------+--------------+--------+----------------------------
| id | select_type | table        | type   | possible_keys               | key
+----+-------------+--------------+--------+----------------------------+
|  1 | SIMPLE      | books        | ALL    | NULL                        | NULL
|  1 | SIMPLE      | authors_books | ref    | authors_books_book_id_index | ➥
authors_books_book_id_index
|  1 | SIMPLE      | authors       | eq_ref | PRIMARY                     | ➥
PRIMARY
|  1 | SIMPLE      | publishers    | eq_ref | PRIMARY                     | ➥
PRIMARY
+----+-------------+--------------+--------+----------------------------+
4 rows in set (0.01 sec)
```

In the key field, you can see that the index used is `authors_books_book_id_index`, the index we just created with the migration.

■ **Tip** For more information about using `EXPLAIN` in MySQL, see `http://dev.mysql.com/doc/refman/5.0/en/explain.html`. For details on the PostgreSQL version, see `www.postgresql.org/docs/8.1/interactive/sql-explain.html`.

As a rule of thumb, you should create indices for all foreign key fields, as well as for all columns you are using in `WHERE` clauses. A good example of such a column is `tags.name`, which is used extensively by the `acts_as_taggable` code to find tags in the database table.

Summary

In this chapter, we took a look at how to measure and optimize the performance of a Rails application. We introduced the Rails Analyzer set of profiling tools. Then we added different caching mechanisms to our application, for entire pages and content chunks that are either needed very often or that take expensive operations to be produced. Finally, we took a look at common performance problems in Rails applications and how to solve them.

Measuring and optimizing a web application are constant processes. It is not enough to make the application fast once and then lean back and enjoy the profits. User behavior changes over time, and the hit rates go up (hopefully), so it pays to follow closely how the site behaves.

Optimizing a Rails application is a topic that merits a book or three just for itself. We could only scratch the surface in this chapter. For a lot more information on this topic, follow Stefan Kaes's blog at `http://railsexpress.de/blog/`, and look for his upcoming book on the topic, *Performance Rails: Building Rails Applications with Sustainable Performance* (ISBN: 0-32147-741-3).

Index

B

C

D

E

F

G

H

I

M

R

S

T

U

V

W

X

Y

Z