# 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**

**Copyright © 2006 by Christian Hellsten and Jarkko Laine**

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.

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.

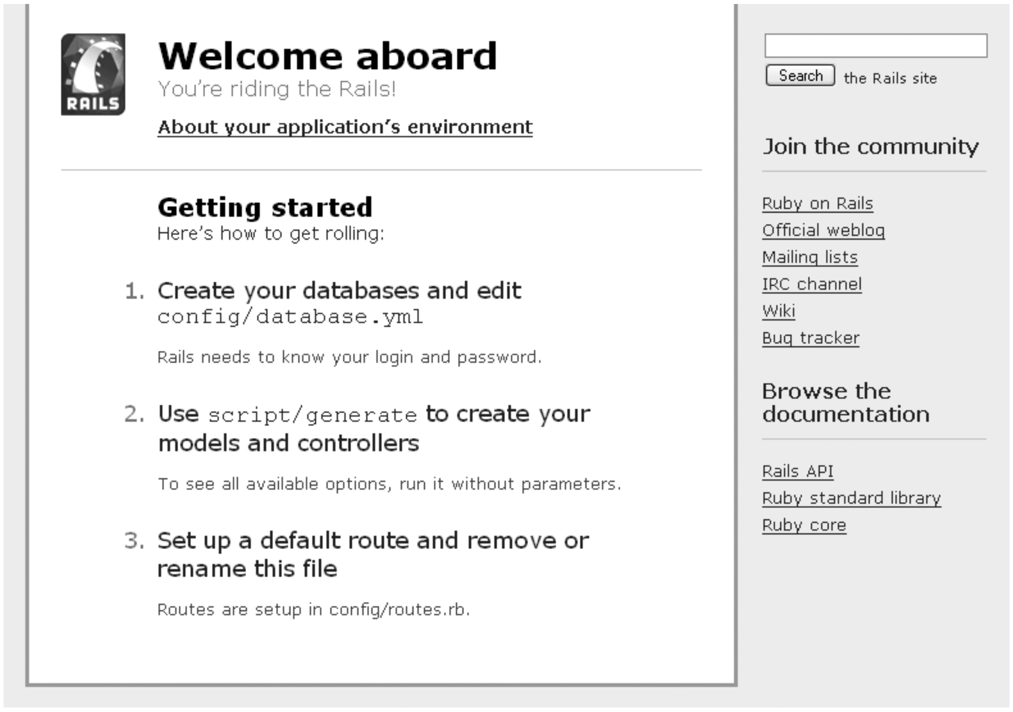
```
$ cd /home/george/projects/emporium
$ script/server
```

```
=> 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`



Welcome aboard
You're riding the Rails!
About your application's environment

Getting started
Here's how to get rolling:

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

   Rails needs to know your login and password.

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

   To see all available options, run it without parameters.

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

   Routes are setup in config/routes.rb.

**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:

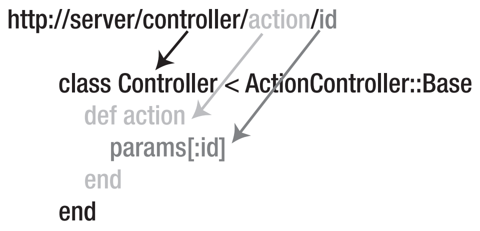
- The *controller* is the component that receives the request from the browser and performs the user-specified action.
- The *model* is the data layer that is used, usually from a controller, to read, insert, update, and delete data stored, for example, in a relational database.
- The *view* is the representation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



```
http://server/controller/action/id

    class Controller < ActionController::Base
      def action
        params[:id]
      end
    end
```

**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.

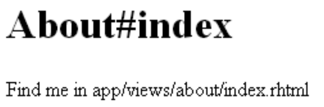
Run the generate script with the following parameters to create the about controller, index action, and related files.

```
$ 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.


About#index

Find me in app/views/about/index.rhtml

**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 `&amp;&lt;&gt;` to the browser, instead of `&<>`, by using the `h` method, which is short for `html_escape`.

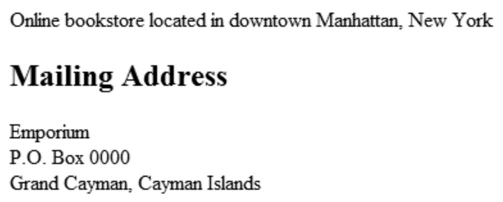
```
<%= h('&<>') %>
```

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.

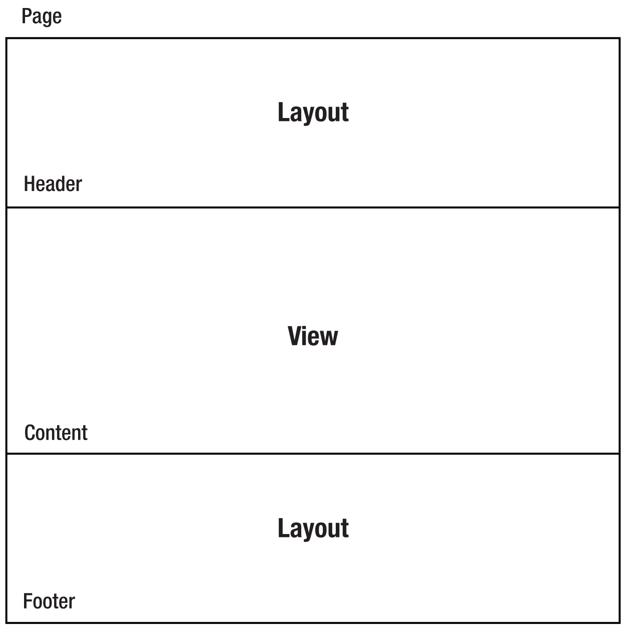


**Figure 1-5.** *Layouts and views*

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.

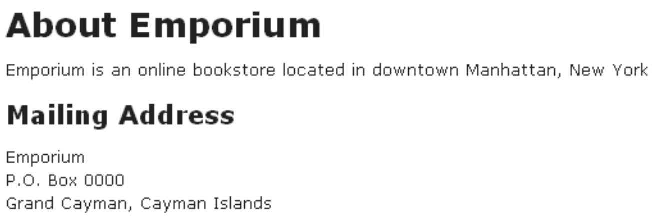
## Modifying the Generated Controller

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.


About Emporium

Emporium is an online bookstore located in downtown Manhattan, New York

Mailing Address

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

**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.