lesson 2

Note

Lesson 1, "Introduction," appears in the video only.

Getting Started

This Live Lesson video series is adapted from Michael Hartl and Aurelius Prochazka's *RailsSpace: Building a Social Networking Website with Ruby on Rails*. The first chapter of the book is available as a free download from the book's website at

http://railsspace.com/book/

When we provide code listings in this booklet, we still adhere to the book's numbering convention because the book website has an archive of all code developed in the book and therefore all the code used in the videos.

It's time to start building RailsSpace, a social networking website for Ruby on Rails enthusiasts (and whoever else shows up). Eventually, RailsSpace will have many of the features associated with popular social networks such as Facebook and MySpace, with searchable user profiles, an email system, and friends lists. The first part of the lesson series lays a foundation by developing a system for registration, login, and authentication. This system is essential for our purposes, but it's also needed for virtually any user-based web application. In the second part, we build a social network on this foundation. Of course, in the process of making this specific type of application, we develop many general techniques useful for building other kinds of websites as well.

In this lesson, we start with the application by making the front page of our site, together with a couple of other static pages. Much of what we do in this lesson could be done quite easily with plain HTML files, but even in this extremely simple case, Rails still proves surprisingly convenient. It also makes for a gentle introduction to some of the core concepts behind Rails, including embedded Ruby and the model-view-controller (MVC) architecture.

Preliminaries

Choosing a Platform

Although we suppose there are alternatives, everyone we know uses either Windows, Macintosh, or Linux for Rails development. Your choice of platform will probably be dictated more by what you are currently familiar with than anything else. In RailsSpace, we strive to support all three of these platforms. Because we like the Apple look, all screencasts are done on the Macintosh platform, but we can report from personal experience that Rails is unambiguously cross-platform; you can build great Rails applications no matter which operating system you choose.

We bet that many of you have, in your excitement, already installed Rails, but if you haven't (or if you're using an older version), you should do that now. After you've chosen a platform (see the sidebar "Choosing a Platform"), head over to the Ruby on Rails download page (http://www.rubyonrails.org/down) for instructions on how to install Rails. There are many different ways to get rolling with Rails; here is one basic sequence:

1. Install Ruby.
   - **Windows:** Download the Windows installer (the first `.exe` file at http://rubyforge.org/frs/?group id=167) and double-click on it.
   - **Linux:** Download the Ruby source code from http://www.ruby-lang.org/ en/downloads/. Linux users, you know the drill: extract with

     ```
     $ tar zxf <filename>
     ```

     and install with

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

     If you don't have `sudo` enabled, you have to log in as root for the final step:

     ```
     $ ./configure
     $ make
     $su
     # make install
     ```

   - **OS X:** The Ruby that ships with OS X 10.4 has some issues, so you might want to look at this: http://hivelogic.com/narrative/articles/ruby_rails_lighttpd_mysql_tiger.
2. Install RubyGems, the standard Ruby package manager.
   - **Windows:** Download the first RubyGems `.zip` file from http://rubyforge.org/frs/?group id=126 and unzip it, extracting the files to a directory on your local disk. Using a command prompt (DOS window), navigate to the directory where you extracted the files and run

     ```
     > ruby setup.rb
     ```

   - **Linux and OS X:** Download the first RubyGems `.tgz` file from http://rubyforge.org/frs/?group id=126, extract it, and run

     ```
     $ ruby setup.rb
     ```

     inside the source directory.
3. Install Rails at the command line:

   ```
   > gem install rails --include-dependencies
   ```

   Now go get a cup of coffee while Rails and all its associated files are automagically installed.

Setting Up Your Development Environment

Your specific development environment will depend somewhat on the platform you choose, but because Rails applications are written in Ruby, at the very least you need a text editor for writing source code. As you'll see later, Rails projects have a lot of different files and directories, so it's useful to have an editor or integrated development environment (IDE) able to navigate the directory tree and switch between files quickly.

We particularly recommend RadRails, a free (as in beer and speech) cross-platform Rails IDE based on Eclipse (which will be familiar to many Java developers out there). For Rails developers working on OS X, the most popular choice seems to be TextMate, a text editor with lots of nice Rails macros and good directory navigation.

Easy as 1-2-3

Rails uses the model-view-controller architecture (see Figure 2.1). Models contain "business logic," including representations of the objects (users, personal data, friendships, and so on) used by your web application; models are responsible for communicating with the back-end data store, typically a relational database such as MySQL. Views are responsible for what the user actually sees; they typically contain a mix of raw HTML and an embedded template language for generating dynamic content. Controllers are responsible for figuring out what to do with user input: They handle incoming browser requests, call the appropriate functions on model objects if necessary, and—through the *actions* that live inside controllers—render views into pure HTML for return to the browser. Together, these three components make for a natural division of labor in web applications.

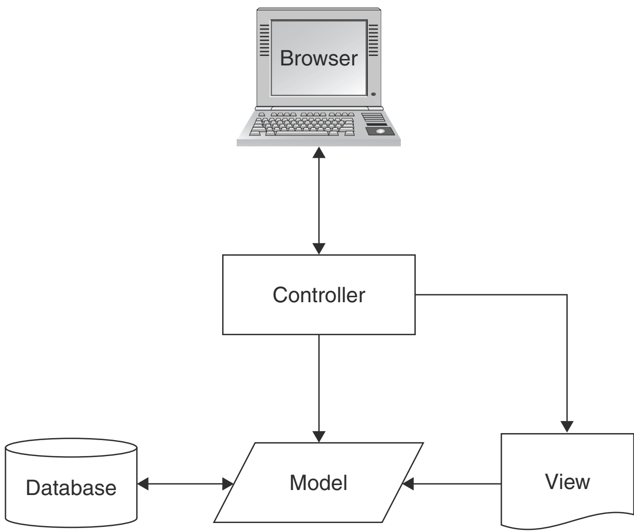

Figure 2.1 Simple representation of the MVC architecture.

The Rails API

The entire Rails application programming interface (API) is available online at http://api.rubyonrails.org/.

When programming Rails, you may find that (like us) you consult the API frequently. To search for something in the API (such as `ActionController::Base`), we usually just use the "find" feature of our browsers.

By the way, if you want to have the Rails API and other documentation available on your local machine, install the docs using

```
> gem rdoc --all
```

and then start the gem documentation server using

```
> gemserver
```

Now all the documentation is available on your local machine on port 8808:

```
http://localhost:8808
```

Hashes

A hash (short for *hash table*), also called an *associative array* or a *dictionary*, is a data type consisting of key-value pairs; you can think of a hash as a sort of generalized array, with the index type not limited to integers. Probably the best way to understand hashes is to look at a few examples using the interactive Ruby program `irb`. Let's start with an empty hash, try to access a nonexistent element, and then add several key-value pairs:

```
> irb
irb(main):001:0> h = {}
=> {}
irb(main):002:0> h["foo"]
=> nil
irb(main):003:0> h["foo"] = "bar"
=> "bar"
irb(main):004:0> h["foo"]
=> "bar"
irb(main):005:0> h["baz"] = "quux"
=> "quux"
irb(main):006:0> h[17] = 123.5
=> 123.5
irb(main):007:0> h
=> {17=>123.5, "baz"=>"quux", "foo"=>"bar"}
```

Note that we access a hash value by putting the key in square brackets; if the hash doesn't have a value corresponding to a particular key, it returns `nil`, which is a special Ruby value for "nothing at all." We can also add elements to the hash using square brackets, as in `h["foo"] = "bar"`. If we type the name of the hash (`h` in this case), `irb` prints the key-value pairs. Ruby hashes can contain multiple types; you probably recognize strings, integers, and floating-point numbers in the preceding examples. Also note that hashes have no intrinsic sense of order, so your version of `irb` might print out the key-value pairs in a different order.

Always having to build up hashes using the bracket notation would be cumbersome, so Ruby lets us define a hash explicitly using curly braces and key-value pairs as follows:

```
irb(main):008:0> h = { :action => "help", :controller => "site"}
=> {:action=>"help", :controller=>"site"}
```

Each key in this example is a *symbol*, which may be unfamiliar to you because few languages have this type. Of course, it's just our luck that this is precisely the syntax that occurs in `link_to`.

Symbols

A Ruby symbol is just a label, formed in the same way as a string, except with a single colon instead of quotation marks. Most languages use strings as labels, especially in hashes, but a string has a lot of properties that have nothing to do with its role as a label. You can find a string's length, access a substring, compare it to a regular expression, reverse it, and so on. Ruby takes the next logical step and creates a separate data type for labels; you can think of symbols as strings without all those other unnecessary attributes. As a result, you can compare symbols very quickly, whereas comparing strings requires that you walk down both strings character by character. Efficient comparison makes symbols ideal for use as hash keys.