

Rails Solutions

Ruby on Rails Made Easy

Justin Williams



Rails Solutions: Ruby on Rails Made Easy

Copyright © 2007 by Justin Williams

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-752-1

ISBN-10 (pbk): 1-59059-752-4

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

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 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 www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor **Copy Editor**
Chris Mills Nancy Sixsmith

Technical Reviewers **Assistant Production Director**
Ashish Bansal Kari Brooks-Copony
Ryan J. Bonnell

Editorial Board **Production Editor**
Steve Anglin Katie Stence

Ewan Buckingham **Composer**
Gary Cornell Molly Sharp
Jason Gilmore

Jonathan Gennick **Artist**
Jonathan Hassell April Milne
James Huddleston

Chris Mills **Proofreader**
Matthew Moodie Linda Seifert
Dominic Shakeshaft

Jim Sumser **Indexer**
Keir Thomas Michael Brinkman
Matt Wade

Project Manager **Interior and Cover Designer**
Beth Christmas Kurt Krames

Copy Edit Manager **Manufacturing Director**
Nicole Flores Tom Debolski

4 GETTING STARTED WITH RAILS

RAILS SOLUTIONS: RAILS MADE EASY

Now it's time to get your hands dirty and create your first Rails application. This chapter focuses on the basics of using Ruby on Rails: creating projects, adding models and controllers to your application, and writing some basic code that gets your application off the ground and running. Before you begin, let's go over what exactly you will be building.

Throughout the course of the book, you will be spending the majority of your time working on a classified ad application similar to craigslist. The application, cleverly titled railslist, will enable users to post their classified ads on the site, assign them to a category, add photos, and create their own user accounts to track their listings. Users will also be able to search through listings to find what they are looking for. The architecture of the application is shown in Figure 4-1.

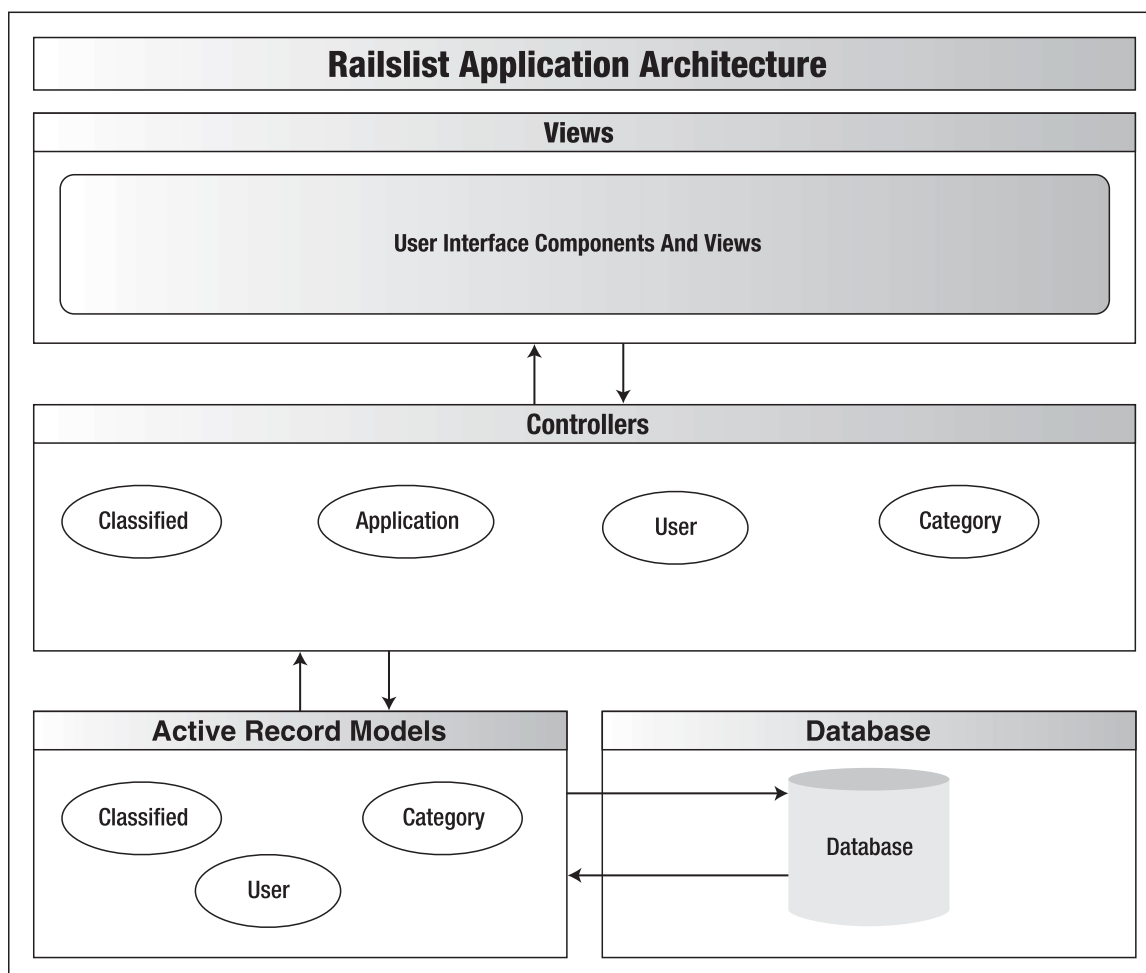


Figure 4-1. The railslist architecture

The application has a basic architecture and will be built using three ActiveRecord models to describe the types of data that is stored:

- Classified, which describes an actual listing
- Category, which is used to group classified ads together
- User, which is used for user accounts

Each time you create an ActiveRecord model object, it is built from a row in the MySQL database. You need to be concerned only with creating the database fields and assigning some validation rules to the models; ActiveRecord handles all the heavy lifting.

Besides the models, there are three controllers—User, Category, and Classified—which enable you to work with each of the model objects. You could use a single controller for the application, but it would not follow the Model-View-Controller (MVC) paradigm well.

The Classified controller enables you to perform the basic create, read, update, delete (CRUD) functionality on your classified ads and enables users to contact the seller to purchase an item. The Category controller enables you to manipulate categories that you can then associate with classified ads. Finally, you have the User controller that enables users to sign up for accounts and then log in to that account. The controllers are where you will write most of your Ruby code (you got the foundation you need to do that in the previous chapter).

The user front-end will be the views that you create for each of the actions in the controller. In terms of the actual users of the railslist application, the front-end is the only thing they are concerned with.

Luckily, Rails makes everything you want to do incredibly easy to accomplish.

The goal of building railslist is to introduce you to as much of the Ruby on Rails framework as possible. As you work through the book, my goal is to show you how easy it is to iteratively develop an application from something very basic into something that can easily be used by anyone around the Web.

There are a few basic steps that are followed each time you create a new Rails application:

1. Use the rails command to create the basic skeleton of the application.
2. Create a database on the MySQL server to hold your data.
3. Configure the application to know where your database is located and the login credentials for it.
4. Start the web server inside the Rails application.
5. Build and test the application.

I'll discuss more of the internals of the application as you proceed through the book.

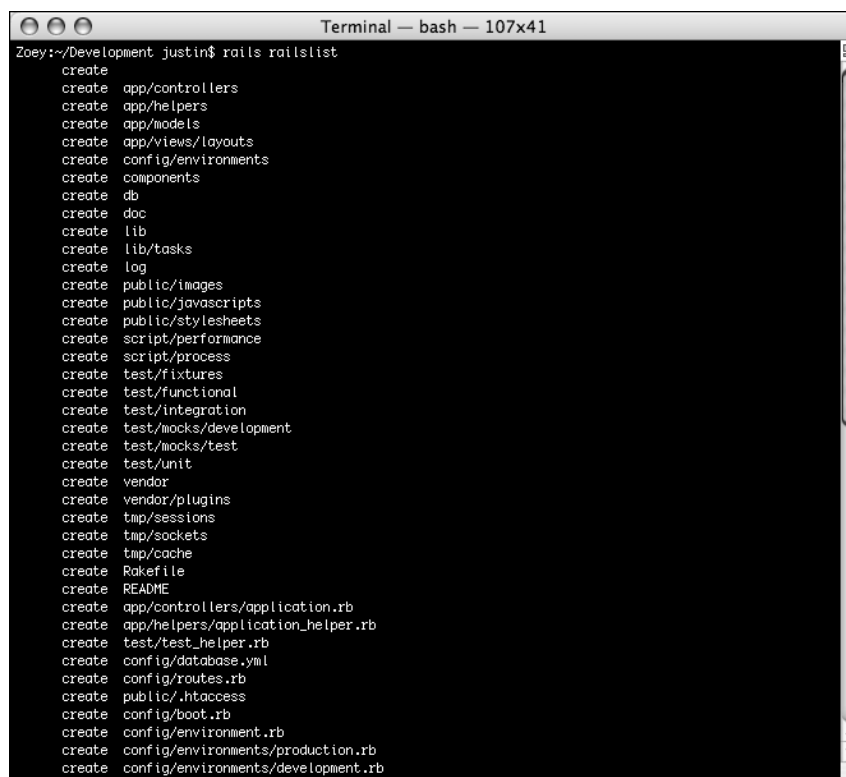
You'll be writing a lot of code in this chapter and subsequent chapters. If you aren't too keen on all that typing, you can visit this book's website and download all the sample code. There's a separate folder for each chapter that contains the completed code as it stands at the end of that chapter.

Let's get started.

Creating a Rails project

The first task that any Rails developer has to do when starting a new project is create the application, which can be done by using one of the many command-line tools available when the framework is installed. The command-line tools can be used with the Mac OS X Terminal application (found in /Application/Utilities/) or the Windows command prompt (Start ► Run ► cmd).

1. The rails command is used to create the skeleton of a new Rails application. Open up a new Terminal or command prompt window and navigate to the directory in which you want to store your application. It doesn't matter where it goes, but I suggest somewhere in the home folder for Mac users and at the root of the c:\ drive for Windows users.
2. After you select a directory, type rails railsgist at the command prompt and press Enter. Your output should look similar to Figure 4-2.
3. Type cd railsgist and press Enter.
4. Type ls on Mac OS X or dir on Windows.

A screenshot of a terminal window titled "Terminal — bash — 107x41". The prompt is "Zoey:~/Development justin\$ rails railsgist". The output lists the files and directories created by the rails command, including app/controllers, app/helpers, app/models, app/views/layouts, config/environments, components, db, doc, lib, lib/tasks, log, public/images, public/javascripts, public/stylesheets, script/performance, script/process, test/fixtures, test/functional, test/integration, test/mocks/development, test/mocks/test, test/unit, vendor, vendor/plugins, tmp/sessions, tmp/sockets, tmp/cache, Rakefile, README, and various application-specific files like application.rb, application_helper.rb, test_helper.rb, config/database.yml, config/routes.rb, public/.htaccess, config/boot.rb, config/environment.rb, config/environments/production.rb, and config/environments/development.rb.

```
Terminal — bash — 107x41
Zoey:~/Development justin$ rails railsgist
create
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  components
create  db
create  doc
create  lib
create  lib/tasks
create  log
create  public/images
create  public/javascripts
create  public/stylesheets
create  script/performance
create  script/process
create  test/fixtures
create  test/functional
create  test/integration
create  test/mocks/development
create  test/mocks/test
create  test/unit
create  vendor
create  vendor/plugins
create  tmp/sessions
create  tmp/sockets
create  tmp/cache
create  Rakefile
create  README
create  app/controllers/application.rb
create  app/helpers/application_helper.rb
create  test/test_helper.rb
create  config/database.yml
create  config/routes.rb
create  public/.htaccess
create  config/boot.rb
create  config/environment.rb
create  config/environments/production.rb
create  config/environments/development.rb
```

Figure 4-2. The output when you use the rails command to create a new application skeleton

If you're using Locomotive or InstantRails to work through this book, all you need to do is use walk through each application's specific project creation wizards to create your new application. You can also skip the forthcoming "Configuring the web server" section because your web server is built in.

The rails command created a lot of directories that are a part of the application. Let's go through each directory and define its purpose.

- **app**: Home to all MVC code.
- **components**: Miniapplications that can bundle controllers, models, and views together. (This subject is covered in Chapter 13.)
- **config**: Database configuration, routing configuration, and environment settings.
- **db**: Database schema files and Rails migration files.
- **doc**: Documentation for an application.
- **lib**: Application-specific custom code that doesn't belong in controllers, models, or helpers (for instance, background processes that work in conjunction with an application are put here). For example, if you were running a stock market tracker, you could write a background process that would ping your stock quote provider for data and put it in this directory.
- **log**: Error and access log files for an application.
- **public**: Cascading Style Sheets (CSS), JavaScript, and other static files.
- **script**: Generator scripts, debugging tools, and performance utilities.
- **test**: Files for testing an application, including unit, fixture, and integration test code. (This subject is covered in Chapter 5.)
- **tmp**: Holds cache files, session information, and socket files used by the web server.
- **vendor**: Where Rails plug-ins are installed. (This subject is covered in Chapter 13.)

Configuring the web server

Rails bundles a web server called WEBrick in the `script` folder, which makes the barrier to entry as low as possible for any platform. Since WEBrick is built using Ruby, anyone who has the Ruby language installed (as shown in Chapter 2) can run it. Included in the `script` directory is a tool called `server`. By default, `server` launches the WEBrick web server that is bundled with Ruby, but if it detects Lighttpd, it instead creates a default `lighttpd` configuration file and uses Lighttpd instead of WEBrick.

If you're a Mac user, you might ask why I had you go through the process of installing Lighty if Rails is bundled with a web server. The reasoning is that I want you to have experience with a production web server such as Lighttpd when developing your applications so that you can have your development environment as close to production quality as possible. Since a majority of Rails applications are deployed using Lighttpd, it only makes sense to show you how to use it in conjunction with developing with Ruby on Rails.

Unfortunately, it is not yet easy enough to configure Lighttpd to work with Windows and Ruby on Rails, so I recommend using WEBrick if you are developing on that platform. The Rails code you write works the same way in both environments.

That said, to launch WEBrick or Lighttpd, go back to the Terminal or command prompt window you used to create your Rails application and type `ruby script/server` and execute the command. You should get an output that looks like this:

```
Zoey:~/railslist justin$ ruby script/server
=> Booting lighttpd (use 'script/server webrick' to force WEBrick)
=> Rails application started on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server (see config/lighttpd.conf for options)
```

What the server just did was create a basic `lighttpd.conf` file in the application's config directory and then launch `lighttpd` using that file if you are using `Lighttpd`. If you are on Windows, `WEBrick` was launched instead. There is no configuration file for it since it is a fairly basic (yet functional) browser.

You will use the `lighttpd.conf` file in Chapter 13 when you deploy the application to a production server.

Viewing the application

Open up a web browser and go to `http://localhost:3000`. You should see a Rails welcome screen like the one shown in Figure 4-3.



Figure 4-3. Default Rails application page

This is the default page for your Rails application. It gives you some pointers on how to get started in developing your Rails application and some links for documentation and support. More importantly, the page shows that your server is working properly. Let's follow its suggestions on how to get started and create the MySQL database.

Creating the database

Your database server should still be running if you are running either Mac OS X or Windows. If you aren't sure that it is running, you can check it via the following methods:

- **Windows:** Open the Windows Control Panel and go to Administrative Tools. Open the Services application and find the MySQL service. If it does not say it is started, double-click it and push the Start button.
- **Mac OS X:** Open the System Preferences application and go to the MySQL preference pane. Click the Start MySQL Server button if it does not already say that the MySQL server instance is running.

RAILS SOLUTIONS: RAILS MADE EASY

When working with Rails, you need to define a separate database for each environment in which you run the application. In this case, it is three environments: development, test, and production. The development database is what you will work with most of the time, but having a production copy on the local machine can be beneficial if you want to test how the application works in a simulated production environment. The test database will be used by Rails' testing framework (covered in Appendix B).

The easiest way to create and manipulate the databases is by using the graphical user interface (GUI) tools you learned about in Chapter 2. For Mac users, it is CocoaMySQL (<http://cocoamysql.sourceforge.net/>); for Windows users, it is SQLyog (<http://www.sqlyog.com/>). If you didn't install the applications before, I recommend downloading and installing them now. The installation process is straightforward for both applications. SQLyog has a basic setup.exe file to walk you through the installation, and CocoaMySQL is an easy drag-and-drop install like most other Mac applications.

Windows

To create a database using SQLyog on Windows, launch the application and follow these steps:

1. In the Connect To MySQL window, enter localhost as the MySQL host address.
2. Enter root as the User Name and your MySQL password in the Password field.
3. Click the Test Connection button to ensure your login credentials work.
4. Under the DB menu, select Create Database.
5. Enter railstest_development in the Create Database popup window.
6. Under the Open Session window, double-click the new Rails Development session.

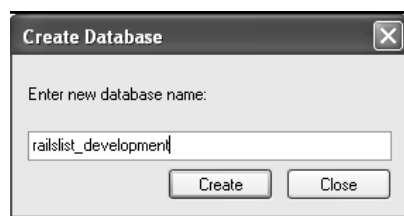


Figure 4-4. Setting the name of the railstest databases on Windows.

Mac OS X

For Mac users using CocoaMySQL, the instructions are similar. After launching CocoaMySQL, follow these steps:

1. In the sheet that pops up, enter localhost as your host, root as your username, and the password to be what you set in Chapter 2. Leave everything else blank so it picks up the default values.
2. Click the Connect button.

- Under Databases in the top-left corner, click the Add database button, as shown in Figure 4-5.

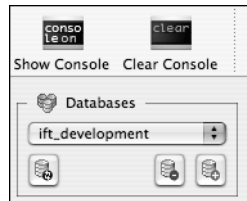


Figure 4-5. Adding a database in CocoaMySQL on Mac OS X.

- A dialog box appears. Type `railslit_development` and click Add.
- Repeat this process two more times, creating the `railslit_test` and `railslit_production` databases, respectively.

Using the command line

If the thought of using GUIs insults your inner geek, you can also create your database using the `mysql` command-line tool.

4

- In Windows, go to Start Menu > All Programs > MySQL > MySQL Server 5.0 > MySQL Command Line Tool (Mac users should just open up a new Terminal window and type `mysql -u root -p`).
- When prompted for your password, enter it.
The MySQL command prompt is not too exciting; it is just a blank screen with `mysql>` preceding it.
- At the prompt, type the following three commands (shown in Figure 4-6):

```
create database railslit_development;
create database railslit_test;
create database railslit_production;
```

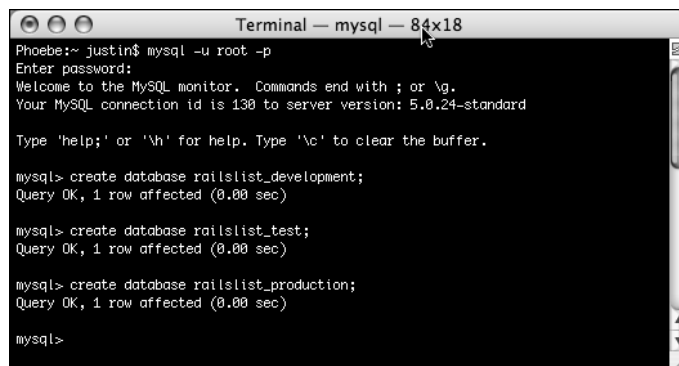


Figure 4-6. Creating the databases by using the command prompt instead is not too difficult.

You just created three blank databases that will be the home of the application data.

You might be wondering about creating the database tables—don't worry about this for now. Later in this chapter, you'll handle this easily using Rails!

Telling Rails about the databases

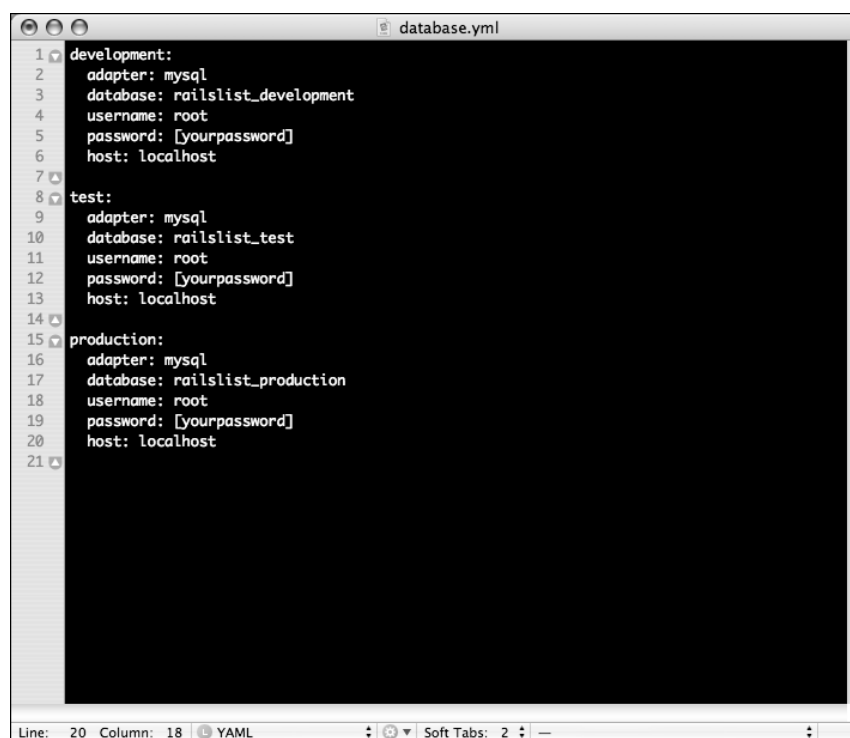
The final step of creating the databases is to tell Rails about them. Database information is stored in the `database.yml` file in the application's config directory.

4. Open the file in your text editor of choice and take a few moments to examine it.

The `database.yml` file is written using YAML. YAML, which stands for `YAML Ain't Another Markup Language`, is used to develop configuration files for scripting languages such as Ruby. Notice that the file is pretty human-readable, and configuration information is stored in key:value pairs. It defines the three execution environments and the database information for each one, respectively.

Since you took some care in the way you named the databases, the only area you need to change in the YAML file is the password for each database.

5. Change all three password fields to contain the MySQL password. When you're done, your file should look similar to Figure 4-7.



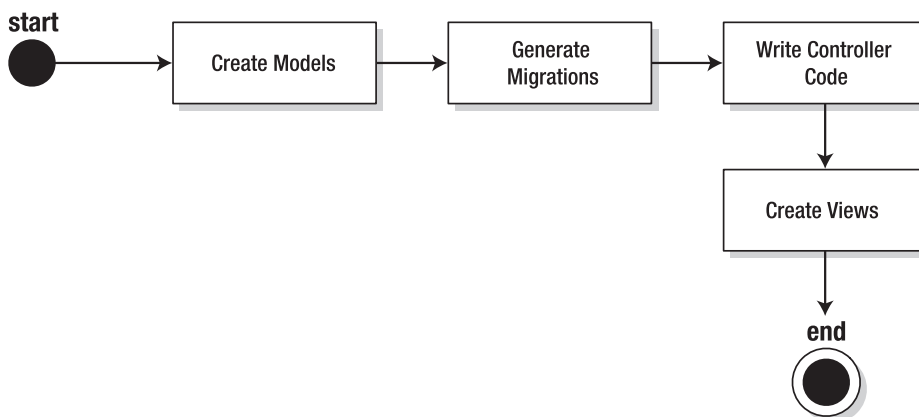
```
1 development:
2   adapter: mysql
3   database: railslist_development
4   username: root
5   password: [yourpassword]
6   host: localhost
7
8 test:
9   adapter: mysql
10  database: railslist_test
11  username: root
12  password: [yourpassword]
13  host: localhost
14
15 production:
16  adapter: mysql
17  database: railslist_production
18  username: root
19  password: [yourpassword]
20  host: localhost
21
```

The screenshot shows a text editor window titled "database.yml". The editor displays the contents of the file, which is a YAML configuration for three environments: development, test, and production. Each environment is defined with a set of key-value pairs: adapter (mysql), database (railslist_*, where * is the environment name), username (root), password ([yourpassword]), and host (localhost). The editor has a dark background and a light-colored border. The status bar at the bottom indicates "Line: 20 Column: 18" and "YAML" is selected. There are also icons for undo, redo, and soft tabs.

Figure 4-7. Updating your YAML file to point to your databases

Creating the model

With the database creation out of the way, you can start focusing on building the application. My recommended workflow (see Figure 4-8) is to first define the model classes because they are the business objects you'll be working with in your controllers. The model also gives you an idea of how to define the fields in your database.



4

Figure 4-8. Recommended workflow for creating Rails applications

To create a model, you simply use the generate model command to create the basic skeleton. Generators, which are a major part of Rails development, enable you to call a single command and perform several tasks at once. Generators are found throughout Rails: creating models, controllers, database migrations, and more.

1. To use the model generator for this example, open up a command prompt or Terminal window, go to the directory in which the application is located, and then type `ruby script/generate model Classified`. You should see the following:

```

Zoey:~ justin$ cd ~/railslist/
Zoey:~/railslist justin$ ruby script/generate model Classified
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/classified.rb
create  test/unit/classified_test.rb
create  test/fixtures/classifieds.yml
create  db/migrate
create  db/migrate/001_create_classifieds.rb
  
```

You're telling the generator to create a model called `Classified` to store instances of classified ads. Each time you create a `Classified` model, you're pulling a row from the database table. Notice that you are capitalizing `Classified` and using the singular form. This is a Rails paradigm that you should follow each time you create a model.

When you use the generate tool, Rails creates the actual model file that holds all the methods unique to the model and the business rules you define, a unit test file for performing test-driven development, a sample data file (called fixtures) to use with the unit tests, and a Rails migration that makes creating database tables and columns easy. (Appendix B covers the unit test and fixtures files.) Right now, let's focus on the model itself and the migrations file.

Rails migrations

Web developers used to have it really hard having to know multiple languages. Besides the basic HTML and CSS knowledge, their tool belts usually included PHP, SQL, and JavaScript as a bare minimum. Rails aims to eliminate two of those languages from the tool belt by making it incredibly easy to write SQL and JavaScript using Ruby. (Rails' solution for complex JavaScript, called RJS, is covered in later chapters.) For now, let's focus on the Rails way of making SQL and database management a snap: migrations.

Migrations were created because the developers of Ruby on Rails realized that data models for an application can change over time and that deploying those changes can sometimes be a difficult task. They also didn't want to have to work with complex SQL queries that could sometimes take line upon line of code.

A migration file contains basic Ruby syntax that describes the data structure of a database table. Let's describe the Classified model using the migration created when you generated it.

2. Go to your railslist directory and open up the 001_create_classifieds.rb file in the db/migrate folder.

If you are using TextMate on Mac OS X, drag your entire railslist folder onto the TextMate icon. A project window displays, which contains all your application's files in a single window. I find this a much easier way to work.

3. After the file is open, look at the first line. Notice that a migration is just another class that inherits from ActiveRecord, which is why you can use many of the luxuries of Rails to manipulate the data structure. A migration file starts with two methods: `self.up` and `self.down`. With Rails you can migrate to specific versions of the data model at any point in time. The code in the `self.up` method is executed when migrating forward while `self.down` is executed when migrating backward (that is, creating a new version of the database or rolling back to a previous version).

An easy way to think about it is that `self.up` is the action you want to perform in the migration file, and `self.down` is the exact opposite. It is just like using the Undo command in Word or some other application—you're just undoing the changes you made. So, for example, if you want to create a table called `classifieds`, you create it in the `self.up` method and then destroy it in `self.down`. Let's look at how you do that.

4. Replace the default code in your `001_create_classifieds.rb` migration file with the following and save your changes:

```
class CreateClassifieds < ActiveRecord::Migration
  def self.up
    create_table :classifieds do |t|
      t.column :title, :string
      t.column :price, :float
      t.column :location, :string
      t.column :description, :text
      t.column :email, :string
      t.column :created_at, :timestamp
      t.column :updated_at, :timestamp
    end
  end

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

4

When you think about a basic classified ad, it contains only a minimal amount of data: a title for the item, a price for the item, a location for the item, a description of the item, and a way to contact the seller. You just created it in this migration. The `self.up` method calls `create_table`, which lets Rails know that it should create this table and then add any columns that are defined between the `create_table` structure.

Rails tables should always be named the pluralized version of your model's name. In the case of the `Classified` model, the table was named `classifieds`. If you had a model called `Food`, you would create a table called `foods`. Rails is even smart enough to know common pluralizations for words like `People`, so it will create a table called `persons`. If you have trouble figuring out the pluralized version of your model, you'll be pleased to know that Geoffrey Grosenbach has created `Pluralizer` to assist you. It can be found at <http://nubyonrails.com/tools/pluralize>.

Migrations support all the basic data types: `:string`, `:text`, `:integer`, `:float`, `:datetime`, `:timestamp`, `:time`, `:date`, `:binary` and `:boolean`:

- `:string` is for small data types such as a classified title
- `:text` is for longer pieces of textual data, such as the description
- `:integer` is for whole numbers
- `:float` is for decimals
- `:datetime` and `:timestamp` store the date and time into a column
- `:date` and `:time` store either the date only or time only
- `:binary` is for storing data such as images, audio, or movies
- `:boolean` is for storing true or false values

You are making use of many of these data types in the table. Code was written to create a column for each of the pieces of data you want to store as well as two special columns: `created_at` and `updated_at`. They are two special database columns that Rails can modify on its own. The `created_at` column is modified only when the row is created with the current time stamp. On the other hand, `updated_at` is modified with the current timestamp each time the row's data is manipulated.

Looking back at the `self.up` method, the `create_table` call is followed by `do |t|`, which enables you to easily define the columns that are a part of this table inside the `create_table` call. The `t` between the goalposts is stuck at the beginning of each column definition, so Rails knows for sure that this column belongs to the `classifieds` table. The basic structure of a column definition is `t.column :column_name, :data_type`.

The `self.down` method is incredibly simple because it has only one line. All it does is remove the `classifieds` table from the database. If you're familiar with basic SQL, it is the same as `drop table classifieds`.

Now that you have created the migration file, you can execute it against the database.

5. To do this, go to a command prompt and go to the `railslist` directory, in which the application is located, and then type `rake migrate`.

```

Zoey:~/railslist justin$ rake migrate
(in /Volumes/Data/Users/justin/railslist)
== CreateClassifieds: migrating =====
-- create_table(:classifieds)
   -> 0.1674s
== CreateClassifieds: migrated (0.1678s)
=====

```

Rake is a Ruby build program similar to the Unix `make` program that Rails takes advantage of to simplify the execution of complex tasks (such as updating a database's structure, for example). Over the course of reading this book, you will become very familiar with executing tasks via Rake.

The database now has a table in which to store the classified ad data, and you didn't have much work to do to accomplish this. Just for comparison, this is what the SQL query looks like if you want to create your table by hand-writing the SQL statement:

```

CREATE TABLE classifieds (
  `id` int(11) DEFAULT NULL auto_increment
  PRIMARY KEY, `title` varchar(255), `price` float,
  `location` varchar(255), `description` text, `email` varchar(255),
  `created_at` datetime, `updated_at` datetime) ENGINE=InnoDB

```

I don't know about you, but I'd rather write a few lines of Ruby than try to match the syntax and data types of that SQL statement.

The next time you want to modify the data model, you can create a new migration file and then run `rake migrate` again. Each time you run the migrate command, it starts at the first migration file (based on the number at the beginning of the filename) and checks to see whether it has been executed. If it has been run, it skips to the next file until it finds a starting point to begin executing. After it finds that point, it runs that migration and all migrations after that until it reaches the end.

Creating the controller

Aside from defining and running the migration, you won't work with the model just yet. Instead, you'll focus on writing the basic code to manipulate the model. That code is stored in a controller class, as you learned in the discussion of MVC in Chapter 1. As outlined before, Ruby on Rails is built using the MVC paradigm, which separates the business objects from the code that manipulates them and hides it all behind a user interface that is visible to your users.

1. To create a controller, open up a command prompt and go to the directory in which the application is located; then type `ruby script/generate controller Classified`.

```

Zoey:~/railslist justin$ ruby script/generate controller Classified
exists  app/controllers/
exists  app/helpers/
create  app/views/classified
exists  test/functional/
create  app/controllers/classified_controller.rb
create  test/functional/classified_controller_test.rb
create  app/helpers/classified_helper.rb

```

Creating controllers is just as easy as models because they both use the generate command-line tool. Besides the controller itself, generate also creates a `classified` folder under `views` that will be where you store the RHTML views (the pages that the user actually sees), a functional test in the `test` folder for test-driven development, and a helper file that interfaces with your views (more on that in future chapters).

2. Let's first take a look at the `classified_controller.rb` file. It is located under `app/controllers`.

```

class ClassifiedController < ApplicationController
end

```

Controller classes inherit from `ApplicationController`, which is the other file in the `controllers` folder: `application.rb`. The `ApplicationController` contains code that can be run in all your controllers and it inherits from Rails' `ActionController::Base` class. You don't need to worry with the `ApplicationController` yet, so let's go back to `classified_controller.rb` and define a few method stubs.

3. Modify the file to look like the following and save your changes:

```
class ClassifiedController < ApplicationController
  def list
  end

  def show
  end

  def new
  end

  def create
  end

  def edit
  end

  def update
  end

  def delete
  end
end
```

These are all the methods that will be a part of the `ClassifiedController`. First, concentrate on the reading methods: `list` and `show`. The `list` method gives you a printout of all the classifieds in the database, while `show` displays only further details on a single classified ad.

4. Modify your code so that the `show` and `list` methods look like the following and then save again:

```
def list
  @classifieds = Classified.find(:all)
end

def show
  @classified = Classified.find(params[:id])
end
```

You added only a single line of code to each method, and that's all you need so far. The `@classifieds = Classified.find(:all)` line in the `list` method tells Rails to search the `classifieds` table and store each row it finds in the `@classifieds` instance object. The `show` method's `@classified = Classified.find(params[:id])` line tells Rails to find only the classified ad that has the `id` defined in `params[:id]`. The `params` object is a container that enables you to pass values between method calls. For example, when you're on the page

called by the `list` method, you can click a link for a specific classified ad, and it passes the id of that ad via the `params` object so `show` can find the specific ad. You can then output that ad's information to the screen (more on this later).

Creating the views

Let's see what happens when you try to execute the `list` method via the web browser.

1. Open up a browser and go to `http://localhost:3000/classified/list`. You'll probably see the message shown in Figure 4-9.

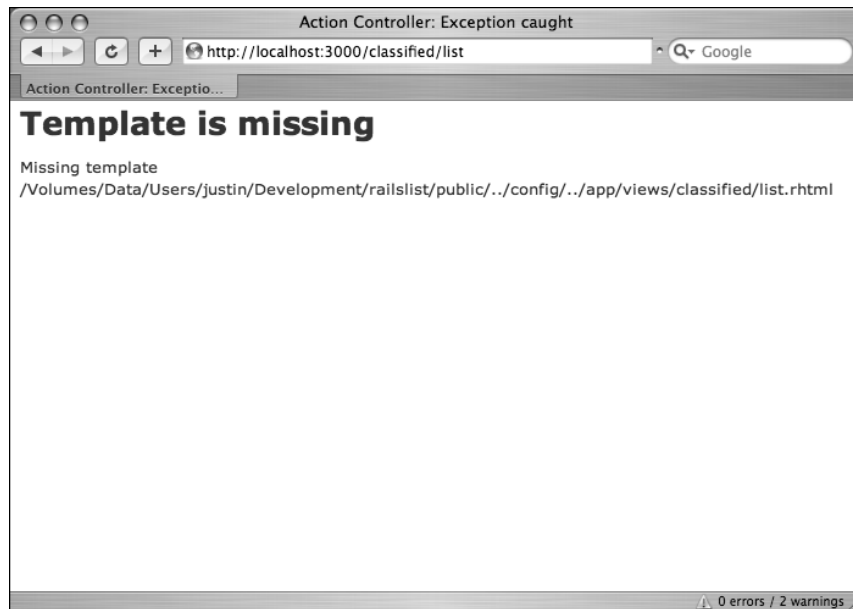


Figure 4-9. You created some application code, but you don't yet have anything to display the data!

Rails lets you know that you need to create the view file for the new method. Each method you define in the controller needs to have a corresponding RHTML file, with the same name as the method, to display the data that the method is collecting. Unfortunately, Rails can't read your mind, so it can't create a view file for each of the controller's methods. It's not a big deal, though. Do the following:

2. Create a file called `list.rhtml` using your favorite text editor and save it to `app/views/classified`.
3. After creating and saving the file, refresh your web browser. You should see a blank page; if you don't, check the spelling of your file and make sure that it is the exactly the same as your controller's method.

4. A blank screen is rather boring, so put some code into the `list.rhtml` file.

```
<% if @classifieds.blank? %>
  <p>There are not any ads currently in the system.</p>
<% else %>
  <p>These are the current classified ads in our system</p>

  <ul id="classifieds">
<% @classifieds.each do |c| %>
  <li><%= link_to c.title, {:action => 'show', :id => c.id} -%></li>
  <% end %>
  </ul>
<% end %>
  <p><%= link_to "Add new ad", {:action => 'new' }%></p>
```

This is a lot to digest, so let's go through it line by line. The first line is enclosed in `<% %>`, which lets Rails know that this is Rails code that should be interpreted. The code to be executed is to check whether the `@classifieds` array has any objects in it. The `.blank?` method returns `true` if the array is empty and `false` if it contains any objects.

The next line outputs a line of HTML if the `@classifieds` array is blank. The third line is a continuation of the line 1 if statement and gives the else clause (that is, if `@classifieds` is *not* blank).

The first two lines after the else clause print some basic HTML tags. After that, things get interesting. You have an each iterator that loops through each item in the `@classifieds` array. Each loop prints out a list item (``) that contains a link to the item.

Notice that the list item line contains `<%= %>` instead of `<% %>`. By appending the `=` sign to the escape clause, you tell Rails that you want to display the output of this Ruby code. The code between the `<%= %>` tags is a `link_to` method call. The first parameter of `link_to` is the text to be displayed between the `<a>` tags. The second parameter is what action is called when the link is clicked. In this case, it is the `show` method. The final parameter is the id of the classified item that is passed via the `params` object.

```

<a href="/new">Add new ad</a>
      ↙         ↘
<%= link_to "Add new ad", {:action => 'new'} %>

```

Figure 4-10.
A link is converted into a standard HTML tag, with the title and href values mapped accordingly.

By using `<%= %>`, Rails puts each output on its own new line, which can cause a bit of clutter in your HTML source. If you are a tidy person, you can use `<%= -%>`, which keeps the code on the same line as the previous line of code.

5. Refresh your browser window; you should see a single line that says there are no ads in the system and an Add new ad link. (This link currently doesn't go anywhere, but you'll be creating the page it targets in the next section, so never fear.) If not, check your code syntax to make sure that everything looks exactly as it does here.

Creating the first objects

Having an application that doesn't have any classifieds is boring, so you need to start populating the application with some real data.

1. Go back to your `classified_controller.rb` file in `app/controllers` and edit the new method to look like this:

```
def new
  @classified = Classified.new
end
```

2. The line you added to the new method lets Rails know that you will create a new object in this view. Create the corresponding `new.rhtml` file in `app/views/classified`.

3. You'll create a basic input form to accept new classified postings. Add the following code to the `new.rhtml` file and save it:

```
<h1>Post new classified</h1>

<%= start_form_tag :action => 'create' %>

  <p><label for="classified_title">Title</label><br/>
  <%= text_field 'classified', 'title' %></p>

  <p><label for="classified_price">Price</label><br/>
  <%= text_field 'classified', 'price' %></p>

  <p><label for="classified_location">Location</label><br/>
  <%= text_field 'classified', 'location' %></p>

  <p><label for="classified_description">Description</label><br/>
  <%= text_area 'classified', 'description' %></p>

  <p><label for="classified_email">Email</label><br/>
  <%= text_field 'classified', 'email' %></p>

  <%= submit_tag "Create" %>
<%= end_form_tag %>

<%= link_to 'Back', {:action => 'list'} %>
```

There are a few new Rails method calls in this template that should be discussed. The first one you will encounter is `start_form_tag()`. This method interprets the Ruby code into a regular HTML `<form>` tag using all the information supplied to it. This tag, for example, outputs the following HTML:

```
<form action="/classified/create" method="post">
```

Two lines below that is a `text_field` method that outputs an `<input>` text field. The parameters for `text_field` are object and field name. In this case, the object is `classified` and the name is `title`. The next new tag you encounter is `submit_tag`, which outputs an `<input>` button that submits the form. Finally, there is the `end_form_tag` method that simply translates into `</form>`.

After creating the form, you need to edit the `create` method so it can take the data submitted by the user and turn it into a row of data in the database.

4. Edit the `create` method in the `classified_controller.rb` to match the following:

```
def create
  @classified = Classified.new(params[:classified])
  if @classified.save
    redirect_to :action => 'list'
  else
    render :action => 'new'
  end
end
```

The first line creates a new instance variable called `@classified` that holds a `Classified` object built from the data the user submitted. The data was passed from the `new` method to `create` using the `params` object (which is why the text fields had their object set to `classified`).

The next line is a conditional that redirects the user to the `list` method if the object saves correctly to the database. If it doesn't save, the user is sent back to the `new` method. The `redirect_to` method is similar to performing a meta refresh on a web page: it automatically forwards you to your destination without any user interaction.

Since the `create` method called a `redirect_to` and `render` method for both of the `if` statement conditionals, you don't need to create a template for the `create` method because it will never have any output on the screen.

5. Go to your browser and visit `http://localhost:3000/classified/new`, enter some data into the form (as seen in Figure 4-11), and submit it.

The screenshot shows a web browser window with the URL `http://localhost:3000/classified/new`. The page title is "Post new classified". The form contains the following fields and content:

- Title:** Model Train Set
- Price:** 450
- Location:** Evansville, IN
- Description:** Train set includes lots of track, many cars, and a caboosse. Cashiers check only. E-mail if interested.
- Email:** justin@secondgearllc.com

At the bottom of the form, there is a "Create" button and a "Back" link. The browser's status bar at the bottom right shows "0 errors / 2 warnings".

4

Figure 4-11. The form to create a new classified ad

Why should you bother learning new methods such as `start_form_tag`, `text_field`, and `submit_tag` instead of just writing straight HTML? Simplicity. Rails has made it very easy to create complex forms more rapidly by simply defining a keyword plus its parameters and then having Rails output the valid HTML for it. Take a look at the source code output by the form you just created and compare it with the code you wrote. You get a lot of payoff for less effort by using Rails' built-in form methods. Unfortunately, not all tags have Rails helpers, which is why you use regular `<label>` tags. Check out <http://www.rubyonrails.org/docs> for more information.

The data should submit successfully and redirect you to the list page, in which you now have a single item listed. If you click the link, you should see another Template is missing error since you haven't created the template file yet.

6. Create a show.rhtml file under app/views/classified and populate it with the following code:

```

<h1><%= @classified.title %></h1>

<p><strong>Price: </strong> $<%= @classified.price %><br />
  <strong>Location: </strong> <%= @classified.location %><br />
  <strong>Date Posted:</strong> <%= @classified.created_at %><br />
  <strong>Last updated:</strong> <%= @classified.updated_at %>
</p>

<p><%= @classified.description %></p>

<hr />

<p>Interested? Contact <%= mail_to @classified.email -%></p>

<%= link_to 'Back', { :action => 'list' } %>

```

There's not much new to this view other than the use of `mail_to` to display the e-mail address. It is similar to `link_to`, but instead creates a `mailto:` link. Also of note is the use of the `created_at` and `updated_at` fields. They are pretty ugly right now, but in later chapters you will do some things to make the display more appealing.

Updating existing ads

The final pieces of the basic implementation of `railslist` include allowing the user to edit and delete listings from the application. Let's tackle editing first.

1. Modify the `classified_controller.rb` edit and update methods to look like the following:

```

def edit
  @classified = Classified.find(params[:id])
end

def update
  @classified = Classified.find(params[:id])
  if @classified.update_attributes(params[:classified])
    redirect_to :action => 'show', :id => @classified
  else
    render :action => 'edit'
  end
end

```

Notice that the `edit` method looks nearly identical to the `show` method. Both methods are used to retrieve a single object based on its `id` and display it on a page. The only difference is that the `show` method is not editable.

The update method has a bit more going on, but it is strikingly similar to the create method you detailed before. The only difference is in line 3 of the method: `if @classified.update_attributes(params[:classified])`. The `update_attributes` method is similar to the `save` method used by `create` but instead of creating a new row in the database, it overwrites the attributes of the existing row (described in the `@classified` object) with the new data provided.

Now let's create the view for the edit method.

2. Create a new file called `edit.rhtml` and save it in `app/views/classified`. Populate it with the following code:

```
<h1>Editing Classified: <%= @classified.title -%></h1>

<%= start_form_tag :action => 'update', :id => @classified %>

  <p><label for="classified_title">Title</label><br/>
  <%= text_field 'classified', 'title' %></p>

  <p><label for="classified_price">Price</label><br/>
  <%= text_field 'classified', 'price' %></p>

  <p><label for="classified_location">Location</label><br/>
  <%= text_field 'classified', 'location' %></p>

  <p><label for="classified_description">Description</label><br/>
  <%= text_area 'classified', 'description' %></p>

  <p><label for="classified_email">Email</label><br/>
  <%= text_field 'classified', 'email' %></p>

  <%= submit_tag "Save changes" %>
<%= end_form_tag %>

<%= link_to 'Back', {:action => 'list' } %>
```

Other than line 1 printing the title of the classified ad and modifying the `start_form_tag` action to be `update` instead of `create` and defining an `id`, it is exactly the same form as the new method. You need to provide the user with an outlet for editing the classifieds, so let's edit the `list.rhtml` file.

3. Go to the `` element and modify it to look like the following:

```
<li>
  <%= link_to c.title, {:action => "show", :id => c.id} -%>
  <small> <%= link_to 'Edit', {:action => "edit",
    :id => c.id} %></small>
</li>
```

All you did was add a link called `Edit` that takes the user to the edit form.

4. Point your browser to `http://localhost:3000/classified/list` and test the new functionality. The list page should now look like Figure 4-12.

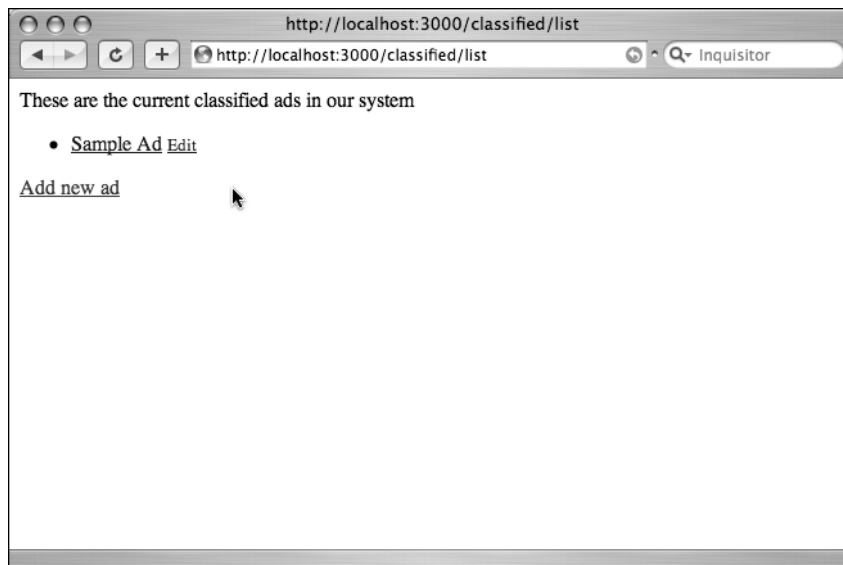


Figure 4-12. The list page, updated with the new edit link

Removing an ad

Removing information from a database using Ruby on Rails is almost too easy. Before you dive into the controller code, let's modify `list.rhtml` again and add a delete link.

1. Go to the `` element and modify it to look like the following:

```
<li>
  <%= link_to c.title, {:action => 'show', :id => c.id} -%>
  <small> <%= link_to 'Edit', {:action => 'edit', :id => c.id} %></small>
  <small> <%= link_to "Delete", {:action => 'delete',
    :id => c.id} %></small>
</li>
```

2. Open `classified_controller.rb` and modify the delete method as follows:

```
def delete
  Classified.find(params[:id]).destroy
  redirect_to :action => 'list'
end
```

The first line finds the classified based on the parameter passed via the `params` object and then deletes it using the `destroy` method. The second line redirects the user to the list method using a `redirect_to` call. This is almost too easy, and you should probably add a confirmation process to protect users against deleting items accidentally. Let's modify `list.rhtml` to confirm the deletions before proceeding.

3. Go back to the `` element and edit it to be like the following:

```
<li>
  <%= link_to c.title, {:action => 'show', :id => c.id} -%>
  <small> <%= link_to 'Edit', {:action => 'edit',
    :id => c.id} %></small>
  <small> <%= link_to "Delete", {:action => 'delete', :id => c.id},
    :confirm => "Are you sure you want to delete this item?" %></small>
</li>
```

The main difference is that you added a `:confirm` parameter that presents a JavaScript confirmation box asking if you really want to perform the action. If the user clicks OK, the action proceeds, and the item is deleted.

Adding some style

Since this is a friends of ED book, and most of us are designers at heart, it is probably making you cringe that there isn't much style on the railslist application. Before wrapping up this chapter, let's work on implementing a layout and some CSS to the application. Ruby on Rails supports the use of layouts for defining a standard layout that is rendered for all actions.

Most websites make use of a layout or templating system. If you look at www.apress.com, as seen in Figure 4-13, you can see that on most pages there is a standard feature set: a logo at the top, navigation bar, and so on. In the main content area, the content changes depending on the book (or set of books) you look at.



Figure 4-13. In this screenshot, the darkened areas are part of the template. The lighter area is changing content per page.

RAILS SOLUTIONS: RAILS MADE EASY

Rails has built-in support for easily adding templating to your applications. The process involves defining a layout template and then letting the controller know that it exists and to use it. First, let's create the template.

1. Add a new file called `standard.rhtml` to `app/views/layouts`. You let the controllers know what template to use by the name of the file, so following a sane naming scheme is advised.
2. Add the following code to the new `standard.rhtml` file and save your changes:

```
<!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>
  <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <meta http-equiv="Content-Language" content="en-us" />
  <title>railslist</title>
<%= stylesheet_link_tag "style" %>
</head>
<body id="rails-list">
<div id="container">
  <div id="header">
    <h1>Railslist</h1>
    <h3>Classifieds powered by Ruby on Rails</h3>
  </div>
  <div id="content">
    <%= yield -%>
  </div>
  <div id="sidebar"></div>
</div>
</body>
</html>
```

Everything you just added were standard HTML elements, except line 7 and 15, which each have a single line of Rails code. Line 7 uses the `stylesheet_link_tag` helper method that outputs a `<link>`. On line 15, the `yield` command lets Rails know that it should put the RHTML for the method called here.

Next, you need to let the Classified controller know about the new template.

3. Open up the `classified_controller.rb` file in `app/controllers` and add the following line just below the first line:

```
layout 'standard'
```

You are telling the controller that you want to use the layout in the `standard.rhtml` file.

4. If you go to `http://localhost:3000/classified/list` you should see that the template is now implemented, as shown in Figure 4-14.

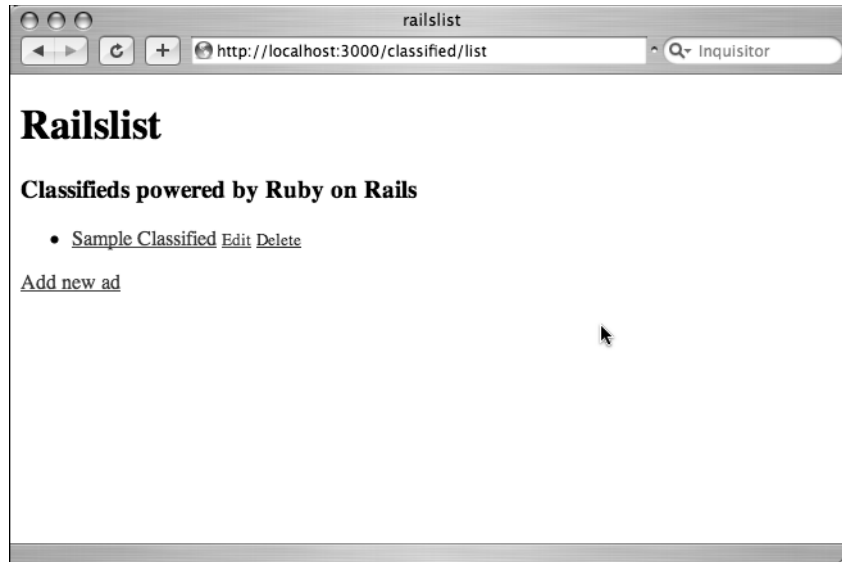


Figure 4-14. The template is now applied to the application.

4

By creating the stylesheet link using the Rails helper method, the fonts were converted to a sans-serif, even though you did not create a stylesheet for the application. This is because Rails defaults to linking to an internal stylesheet when it cannot find the actual CSS file you are referencing. Let's create the CSS file now.

5. Create a new file called `style.css` and save it in `/public/stylesheet`s.

Anything you store in the `/public` directory is viewable by anyone who accesses the application from the web browser. So after you create the `style.css` file, you can access it via the Web at `http://localhost:3000/stylesheet`s/`style.css`.

6. Add the following code to the CSS file and save your changes:

```
* {
  margin: 0;
  padding: 0;
}

body {
  font-family: Helvetica, Geneva, Arial, sans-serif;
  font-size: small;
  font-color: #000;
  background-color: #fff;
}

a:link, a:active, a:visited {
  color: #CD0000;
}
```

RAILS SOLUTIONS: RAILS MADE EASY

```
a:hover {
  color: #F70000;
}

input {
  margin-bottom: 5px;}

p { line-height: 150%; }

div#container {
  width: 760px;
  margin: 0 auto;
}

div#header {
  text-align: center;
  padding-bottom: 15px;
}

div#content {
  float: left;
  width: 450px;
  padding: 10px;
}

div#content h3 {
  margin-top: 15px;
}

ul#classifieds {
  list-style-type: none;
}

ul#classifieds li {
  line-height: 140%;
}

div#sidebar {
  width: 200px;
  margin-left: 480px;
}
```

7. Refresh your browser and you should see your application displayed with a bit more style, as shown in Figure 4-15.

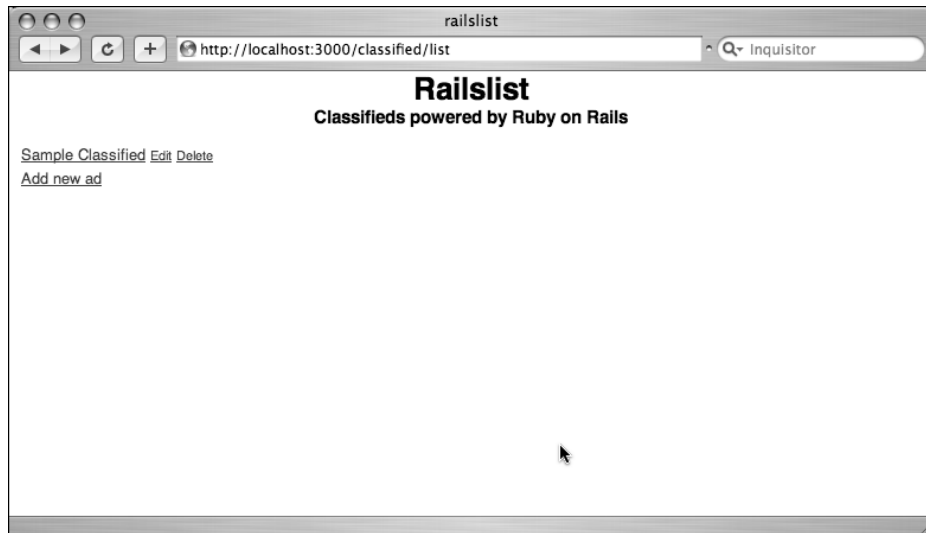


Figure 4-15. railslist in style!

4

Summary

This chapter covered a lot of territory. You started by creating the basics of your first Ruby on Rails application, which gave you a valuable hands-on introduction to the Rails way of doing things. You created your model and database migration, and then added a controller and methods with corresponding views to allow users to view and modify data. Next, you styled the application using a template and a stylesheet.

In the next chapter, you will begin to expand on your Rails knowledge by introducing model validations to the data model and learning some basic debugging skills.