CHAPTER 15
Web Development: Ruby on Rails

Ruby on Rails is unquestionably Ruby's killer app. It can take a lot of credit for lifting Ruby from obscurity outside its native Japan. No other programming language can boast a simple web application framework that also has almost all of that language's developer mindshare.* This chapter demonstrates the principles underlying basic Rails usage (in recipes like Recipe 15.6), gives Rails implementations of common web application patterns (Recipes 15.4 and 15.8) and shows how to use standard Ruby tools from within Rails (Recipes 15.22 and 15.23).

Despite its quality and popularity, Rails does not bring anything new to web development. Its foundations are in standard programming patterns like ActiveRecord and Model-View-Controller. It reuses many preexisting Ruby libraries (like Rake and ERb). The power of Rails is in combining these standard techniques with a ruthless dedication to automating menial tasks, and to asserting resonable default behaviors.

If Rails has a secret, it's the power of naming conventions. The vast majority of web applications are CRUD applications: create, read, update, and delete information from a database. In these types of applications, Rails shines. You start with a database schema and with almost no code, but Rails ties together many pieces with naming conventions and shortcuts. This lets you put meat on your application very quickly.

Because so many settings and names can be sensibly derived from other pieces of information, Rails has much less "paperwork" than other frameworks. Data that's implicit in the code or the database schema doesn't need to be specified anywhere else. An essential part of this system is the ActiveSupport system for pluralizing nouns (Recipe 15.7).

* Python, for instance, has several excellent web application frameworks, but that's just the problem. It has *several*, and a powerful community is fractured on the issue of which to use. Ruby has no major web application frameworks apart from Rails. In a sense, Ruby's former obscurity is what made the dominance of Rails possible.

Where naming conventions can't do the job, Rails uses decorator methods to declare relationships between objects. This happens within the Ruby classes affected by those relationships, not in a bloated XML configuration file. The result is a smaller, simpler to understand, and more flexible application.

As mentioned above, Rails is built on top of common Ruby libraries, and many of them are also covered elsewhere in this book. These libraries include ActiveRecord (much of Chapter 13, but especially Recipe 13.11), ActionMailer (Recipe 14.5), ERb (Recipe 1.3), Rake (Chapter 19), and Test::Unit (Recipe 17.7). Some of these predate Rails, and some were written for Rails but can be used outside of it. The opposite is also true: since a Rails application can be used for many purposes, nearly every recipe in this book is useful within a Rails program.

Rails is available as the `rails` gem, which contains libraries and the `rails` command-line program. This is the program you run to create a Rails application. When you invoke this program (for instance, with `rails mywebapp`), Rails generates a directory structure for your web application, complete with a WEBrick testing server and unit testing framework. When you use the `script/generate` script to jumpstart the creation of your application, Rails will populate this directory structure with more files. The code generated by these scripts is minimal and equivalent to the code generated by most IDEs when starting a project.

The architecture of Rails is the popular Model-View-Controller architecture. This divides the web application into three predictably named parts. We'll cover them in detail throughout this chapter, but here's an introductory reference.

The *model* is a representation of the dataset used by the application. This is usually a set of Ruby classes, subclasses of `ActiveRecord::Base`, each corresponding to a table in the application database. The first serious model in this chapter shows up in Recipe 15.6. To generate a model for a certain database table, invoke `script/generate model` with the name of the table, like so:

```
$ script/generate model users
```

This creates a file called *app/models/users.rb*, which defines a `User` ActiveRecord class as well as the basic structure to unit test that model. It does not create the actual database table.

The *controller* is a Ruby class (a subclass of `ActionController::Base`) whose methods define operations on the model. Each operation is defined as a method of the controller.

To generate a controller, invoke `script/generate controller` with the name of the controller, and the actions you want to expose:

```
$ script/generate controller user add delete login logout
```

This command creates a file `app/controllers/user_controller.rb`, which defines a class `UserController`. The class defines four stub methods: `add`, `delete`, `login`, and `logout`, each corresponding to an action the end user can perform on the objects of

the underlying `User` model. It also creates the template for functionally unit testing your controller.

The controller shows up in the very first recipe of this chapter (Recipe 15.1).

The *view* is the user interface for the application. It's contained in a set of ERb templates, stored in *.rhtml* files. Most importantly, there is usually one *.rhtml* file for each action of each controller: this is the web interface for that particular action. The same command that created the `UserController` class above also created four files in *app/views/user/*: *add.rhtml*, *delete.rhtml*, *login.rhtml*, and *logout.rhtml*. As with the `UserController` class, these start out as stub files; your job is to customize them to present an interface to your application.

Like the controller, the view shows up in the first recipe of this chapter, Recipe 15.1. Recipes like 15.3, 15.5, and 15.14 show how to customize your views.

This division is not arbitrary. If you restrict code that changes the database to the model, it's easy to unit test that code and audit it for security problems. By moving all of your processing code into the controller, you separate the display of the user interface from its internal workings. The most obvious benefit of this is that you can have a UI designer modify your view templates without making them work around a lot of Ruby code.

The best recipes for learning how Model-View-Controller works are Recipe 15.2, which explores the relationship between the controller and the view; and Recipe 15.16, which combines all three.

Here are some more resources for getting started with Rails:

- This book's sister publication, *Rails Cookbook* by Rob Orsini (O'Reilly), covers Rails problems in more detail, as does *Rails Recipes* by Chad Fowler (Pragmatic Programmers)
- *Agile Web Development with Rails* by Dave Thomas, David Hansson, Leon Breedt, Mike Clark, Thomas Fuchs, and Andrea Schwarz (Pragmatic Programmers) is the standard reference for Rails programmers
- The Ruby on Rails web site at *http://www.rubyonrails.com/*, especially the RDoc documentation (*http://api.rubyonrails.org/*) and wiki (*http://wiki.rubyonrails.com/*)

15.1 Writing a Simple Rails Application to Show System Status

Problem

You would like to get started with Rails by building a very simple application.

Solution

This example displays the running processes on a Unix system. If you're developing on Windows, you can substitute some other command (such as the output of a `dir`) or just have your application print a static message.

First, make sure you have the `rails` gem installed.

To create a Rails application, run the `rails` command and pass in the name of your application. Our application will be called "status".

```
$ rails status
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
...
```

A Rails application needs at least two parts: a *controller* and a *view*. Our controller will get information about the system, and our view will display it.

You can generate a controller and the corresponding view with the `generate` script. The following invocation defines a controller and view that implement a single action called `index`. This will be the main (and only) screen of our application.

```
$ cd status
$ ./script/generate controller status index
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/status
      exists  test/functional/
      create  app/controllers/status_controller.rb
      create  test/functional/status_controller_test.rb
      create  app/helpers/status_helper.rb
      create  app/views/status/index.rhtml
```

The generated controller is in the Ruby source file `app/controllers/status_controller.rb`. That file defines a class `StatusController` that implements the `index` action as an empty method called `index`. Fill out the `index` method so that it exposes the objects you want to use in the view:

```
class StatusController < ApplicationController
  def index
    # This variable won't be accessible to the view, since it is local
    # to this method
    time = Time.now

    # These variables will be accessible in the view, since they are
    # instance variables of the StatusController.
    @time = time
    @ps = `ps aux`
  end
end
```

The generated view is in `app/views/status/index.rhtml`. It starts out as a static HTML snippet. Change it to an ERb template that uses the instance variables set in `StatusController#index`:

```
<h1>Processes running at <%= @time %></h1>
<pre><%= @ps %></pre>
```

Now our application is complete. To run it, start up the Rails server with the following command:

```
$ ./script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
...
```

You can see the application by visiting *http://localhost:3000/status/*.

Of course, you wouldn't expose this application to the outside world because it might give an attacker information about your system.

Discussion

The first thing you should notice about a Rails application is that you do not create separate code files for every URL. Rails uses an architecture in which the controller (a Ruby source file) and a view (an ERb template in an `.rhtml` file) team up to serve a number of *actions*. Each action handles some of the URLs on your site.

Consider a URL like *http://www.example.com/hello/world*. To serve that URL in your Rails application, you'd create a `hello` controller and give it an action called `world`.

```
$ ./script/generate controller hello world
```

Your controller class would have a `world` method, and your *views/hello* directory would have a *world.rhtml* file containing the view.

```
class HelloController < ApplicationController
  def world
  end
end
```

Visiting *http://www.example.com/hello/world* would invoke the `HelloController#world` method, interpret the *world.rhtml* template to obtain some HTML output, and serve that output to the client.

The default action for a controller is `index`, just as the default page in a directory of a static web server is *index.html*. So visiting *http://www.example.com/hello/* is the same as visiting *http://www.example.com/hello/index/*.

As mentioned above, a view file is only the main snippet of the final page served by Rails. It's not a full HTML page, and you should never put `<html>` or `<body>` tags inside it (see Recipe 15.3). Since a view file is an ERB template, you should also never

call `puts` or `print` inside a view. ERB was introduced in Recipe 1.3, but it's worth exploring here within the context of a Rails application.

To insert the value of a Ruby expression into an ERB template, use the `<%= %>` directive. Here's a possible *world.rhtml* view for our `hello` action:

```
<p>Several increasingly silly ways of displaying "Hello world!":</p>

<p><%= "Hello world!" %></p>
<p><%= "Hello" + "world!" %></p>
<p><%= w = "world"
       "Hello #{w}!" %></p>
<p><%= 'H' + ?e.chr + ('l' * 2) %><%=('o word!').gsub('d', 'ld')%></p>
```

The last example is excessive, but it proves a point. You shouldn't have to put so much Ruby code in your view template (it should probably go into your controller, or you'll end up with sloppy PHP-like code), but it's possible if you need to do it.

The equals sign in the ERb directive means that the output is to be printed. If you want to execute a command without output, omit the equals sign and use the `<% %>` directive.

```
<% hello = "Hello" %>
<% world = "world!" %>
<%= hello %> <%= world %>
```

A view and a controller may be based on nothing more than some data obtained from within Ruby code (like the current time and the output of `ps aux`). But most real-world views and controllers are based on a *model*: a set of database tables containing data that the view displays and the controller manipulates. This is the famous "Model-View-Controller" architecture, and it's by no means unique to Rails.

See Also

- Recipe 1.3, "Substituting Variables into an Existing String," has more on ERB
- Recipe 15.3, "Creating a Layout for Your Header and Footer"

15.2 Passing Data from the Controller to the View

Problem

You want to pass data between a controller and its views.

Solution

The view is an ERB template that is interpreted within the context of its controller object. A view cannot call any of the controller's methods, but it can access the controller's instance variables. To pass data to the view, set an instance variable of the controller.

Here's a NovelController class, to be put into app/controllers/novel_controller.rb. You can generate stubs for it by running script/generate controller novel index.

```
class NovelController < ApplicationController
  def index
    @title = 'Shattered View: A Novel on Rails'
    one_plus_one = 1 + 1
    increment_counter one_plus_one
  end

  def helper_method
    @help_message = "I see you've come to me for help."
  end

  private

  def increment_counter(by)
    @counter ||= 0
    @counter += by
  end
end
```

Since this is the Novel controller and the index action, the corresponding view is in app/views/novel/index.rhtml.

```
<h1><%= @title %></h1>

<p>I looked up, but saw only the number <%= @counter %>.</p>

<p>"What are you doing here?" I asked sharply. "Was it <%=
@counter.succ %> who sent you?"</p>
```

The view is interpreted after NovelController#index is run. Here's what the view can and can't access:

- It can access the instance variables @title and @counter, because they've been defined on the NovelController object by the time NovelController#index finishes running.
- It can call instance methods of the instance variables @title and @counter.
- It cannot access the instance variable @help_message, because that variable is defined by the method helper_method, which never gets called.
- It cannot access the variable one_plus_one, because that's not an instance variable: it's local to the index method.
- Even though it runs in the context of NovelController, it cannot call any method of NovelController—neither helper_method nor set_another_variable. Nor can it call index again.

Discussion

The action method of a controller is responsible for creating and storing (in instance variables) all the objects the view will need to do its job. These variables might be as simple as strings, or they might be complex helper classes. Either way, most of your application's logic should be in the controller. It's okay to do things in the view like iterate over data structures, but most of the work should happen in the controller or in one of the objects it exposes through an instance variable.

Rails instantiates a new `NovelController` object for every request. This means you can't persist data between requests by putting it in controller instance variables. No matter how many times you reload the page, the `@counter` variable will never be more than two. Every time `increment_counter` is called, it's called on a brand new `NovelController` object.

Like any Ruby class, a Rails controller can define class variables and constants, but they will not be available to the view. Consider a `NovelController` that looks like this:

```
class NovelController < ApplicationController
  @@numbers = [1, 2, 3]
  TITLE = 'Revenge of the Counting Numbers'
end
```

Neither `@@numbers` nor `TITLE` are accessible from within any of this controller's views. They can only be used by the controller methods.

However, contants defined outside of the context of a controller are accessible to every view. This is useful if you want to declare the web site's name in one easy-to-change location. The `config/environment.rb` file is a good place to define these constants:

```
# config/environment.rb
AUTHOR = 'Lucas Carlson'
...
```

It is almost always a bad idea to use global variables in object-oriented programming. But Ruby does have them, and a global variable will be available to any view once it's been defined. They will be universally available whether they were defined within the scope of the action, the controller, or outside of any scope.

```
$one = 1
class NovelController < ApplicationController
  $two = 2
  def sequel
    $three = 3
  end
end
```

Here's a view, `sequel.rhtml`, that uses those three global variables:

```
Here they come, the counting numbers, <%= $one %>, <%= $two %>, <%= $three %>.
```

15.3 Creating a Layout for Your Header and Footer

Problem

You want to create a header and footer for every page on your web application. Certain pages should have special headers and footers, and you may want to dynamically determine which header and footer to use for a given request.

Solution

Many web applications let you define header and footer files, and automatically include those files at the top and bottom of every page. Rails inverts this pattern. A single file called contains both the header and footer, and the contents of each particular page are inserted into this file.

To apply a layout to every page in your web application, create a file called *app/views/layouts/application.rhtml*. It should look something like this:

```
<html>
  <head>
    <title>My Website</title>
 </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

The key piece of information in any layout file is the directive `<%= content_for_layout %>`. This is replaced by the content of each individual page.

You can make customized layouts for each controller independently by creating files in the *app/views/layouts* folder. For example, *app/views/layouts/status.rhtml* is the layout for the `status` controller, `StatusController`. The layout file for `PriceController` would be *price.rhtml*.

Customized layouts override the site-wide layout; they don't add to it.

Discussion

Just like your main view templates, your layout templates have access to all the instance variables set by the action. Anything you can do in a view, you can do in a layout template. This means you can do things like set the page title dynamically in the action, and then use it in the layout:

```
class StatusController < ActionController:Base
  def index
    @title = "System Status"
  end
end
```

Now the application.rhtml file can access @title like this:

```
<html>
  <head>
    <title>My Website - <%= @title %></title>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

application.rhtml doesn't just happen to be the default layout template for a Rails application's controllers. It happens this way because every controller inherits from ApplicationController. By default, a layout's name is derived from the name of the controller's class. So ApplicationController turns into application.rhtml. If you had a controller named MyFunkyController, the default filename for the layout would be app/views/layouts/my_funky.rhtml. If that file didn't exist, Rails would look for a layout corresponding to the superclass of MyFunkyController, and find it in app/views/layouts/application.rhtml.

To change a controller's layout file, call its layout method:

```
class FooController < ActionController:Base
  # Force the layout for /foo to be app/views/layouts/bar.rhtml,
  # not app/view/layouts/foo.rhtml.
  layout 'bar'
end
```

If you're using the render method in one of your actions (see Recipe 15.5), you can pass in a :layout argument to render and give that action a different layout from the rest of the controller. In this example, most actions of the FooController use bar.rhtml for their layout, but the count action uses count.rhtml:

```
class FooController < ActionController:Base
  layout 'bar'

  def count
    @data = [1,2,3]
    render :layout => 'count'
  end
end
```

You can even have an action without a layout. This code gives all of FooController's actions a layout of bar.html, except for the count action, which has no layout at all: it's responsible for all of its own HTML.

```
class FooController < ActionController:Base
  layout 'bar', :except => 'count'
end
```

If you need to calculate the layout file dynamically, pass a method symbol into the layout method. This tells layout to call a method on each request; the return value of

this method defines the layout file. The method can call `action_name` to determine the action name of the current request.

```
class FooController < ActionController:Base
  layout :figure_out_layout

  private

  def figure_out_layout
    if action_name =~ /pretty/
      'pretty'          # use pretty.rhtml for the layout
    else
      'standard'        # use standard.rhtml
    end
  end
end
```

Finally, `layout` accepts a lambda function as an argument. This lets you dynamically decide on a layout with less code:

```
class FooController < ActionController:Base
  layout lambda { |controller| controller.logged_in? ? 'user' : 'guest' }
end
```

It's freeing for both the programmer and the designer to use a layout file instead of separate headers and footers: it's easier to see the whole picture. But if you need to use explicit headers and footers, you can. Create files called `app/views/layouts/_header.rhtml` and `app/views/layouts/_footer.rhtml`. The underscores indicate that they are "partials" (see Recipe 15.14). To use them, set your actions up to use no layout at all, and write the following code in your view files:

```
<%= render :partial => 'layouts/header' %>
... your view's content goes here ...
<%= render :partial => 'layouts/footer' %>
```

See Also

- Recipe 15.5, "Displaying Templates with Render"
- Recipe 15.14, "Refactoring the View into Partial Snippets of Views"

15.4 Redirecting to a Different Location

Problem

You want to redirect your user to another of your application's actions, or to an external URL.

Solution

The class `ActionController::Base` (superclass of `ApplicationController`) defines a method called `redirect_to`, which performs an HTTP redirect. To redirect to

another site, you can pass it a URL as a string. To redirect to a different action in your application, pass it a hash that specifies the controller, action, and ID.

Here's a `BureaucracyController` that shuffles incoming requests to and fro between various actions, finally sending the client to an external site:

```
class BureaucracyController < ApplicationController
  def index
    redirect_to :controller => 'bureaucracy', :action => 'reservation_window'
  end

  def reservation_window
    redirect_to :action => 'claim_your_form', :id => 123
  end

  def claim_your_form
    redirect_to :action => 'fill_out_your_form', :id => params[:id]
  end

  def fill_out_your_form
    redirect_to :action => 'form_processing'
  end

  def form_processing
    redirect_to "http://www.dmv.org/"
  end
end
```

If you run the Rails server and hit *http://localhost:3000/bureaucracy/* in your browser, you'll end up at *http://www.dmv.org/*. The Rails server log will show the chain of HTTP requests you made to get there:

```
"GET /bureaucracy HTTP/1.1" 302
"GET /bureaucracy/reservation_window HTTP/1.1" 302
"GET /bureaucracy/claim_your_form/123 HTTP/1.1" 302
"GET /bureaucracy/fill_out_your_form/123 HTTP/1.1" 302
"GET /bureaucracy/form_processing HTTP/1.1" 302
```

You don't need to create view templates for all of these actions, because the body of an HTTP redirect isn't displayed by the web browser.

Discussion

The `redirect_to` method uses smart defaults. If you give it a hash that doesn't specify a controller, it assumes you want to move to another action in the same controller. If you leave out the action, it assumes you are talking about the `index` action.

From the simple redirects given in the Solution, you might think that calling `redirect_to` actually stops the action method in place and does an immediate HTTP redirect. This is not true. The action method continues to run until it ends or you call `return`. The `redirect_to` method doesn't do a redirect: it tells Rails to do a redirect once the action method has finished running.

Here's an illustration of the problem. You might think that the call to redirect_to below prevents the method do_something_dangerous from being called.

```
class DangerController < ApplicationController
  def index
    redirect_to (:action => 'safety') unless params[:i_like_danger]
    do_something_dangerous
  end

  # ...
end
```

But it doesn't. The only way to stop an action method from running all the way to the end is to call return.* What you really want to do is this:

```
class DangerController < ApplicationController
  def index
    redirect_to (:action => 'safety') and return unless params[:i_like_danger]
    do_something_dangerous
  end
end
```

Notice the and return at the end of redirect_to. It's very rare that you'll want to execute code after telling Rails to redirect the user to another page. To avoid problems, make a habit of adding and return at the end of calls to redirect_to or render.

See Also

- The generated RDoc for the methods ApplicationController::Base#redirect_to and ApplicationController::Base#url_for

15.5 Displaying Templates with Render

Problem

Rails's default mapping of one action method to one view template is not flexible enough for you. You want to customize the template that gets rendered for a particular action by calling Rails's rendering code directly.

Solution

Rendering happens in the ActionController::Base#render method. Rails's default behavior is to call render after the action method runs, mapping the action to a corresponding view template. The foo action gets mapped to the foo.rhtml template.

* You could throw an exception, but then your redirect wouldn't happen: the user would see an exception screen instead.

You can call render from within an action method to make Rails render a different template. This controller defines two actions, both of which are rendered using the shopping_list.rhtml template:

```
class ListController < ApplicationController
  def index
    @list = ['papaya', 'polio vaccine']
    render :action => 'shopping_list'
  end

  def shopping_list
    @list = ['cotton balls', 'amino acids', 'pie']
  end
end
```

By default, render assumes that you are talking about the controller and action that are running when render is called. If you call render with no arguments, Rails will work the same way it usually does. But specifying 'shopping_list' as the view overrides this default, and makes the index action use the shopping_list.rhtml template, just like the shopping_list action does.

Discussion

Although they use the same template, visiting the index action is not the same as visiting the shopping_list action. They display different lists, because index defines a different list from shopping_list.

Recall from Recipe 15.4 that the redirect method doesn't perform an immediate HTTP redirect. It tells Rails to do a redirect once the current action method finishes running. Similarly, the render method doesn't do the rendering immediately. It only tells Rails which template to render when the action is complete.

Consider this example:

```
class ListController < ApplicationController
  def index
    render :action => 'shopping_list'
    @budget = 87.50
  end

  def shopping_list
    @list = ['lizard food', 'baking soda']
  end
end
```

You might think that calling index sets @list but not @budget. Actually, the reverse is true. Calling index sets @budget but not @list.

The @budget variable gets set because render does not stop the execution of the current action. Calling render is like sealing a message in an envelope that gets opened by Rails at some point in the future. You're still free to set instance variables and

make other method calls. Once your action method returns, Rails will open the envelope and use the rendering strategy contained within.

The `@list` variable does *not* get set because the `render` call does not call the `shopping_list` action. It just makes the existing action, `index`, use the `shopping_list.rhtml` template instead of the `index.rhtml` template. There doesn't even need to *be* a `shopping_list` action: there just has to be a template named `shopping_list.rhtml`.

If you do want to invoke one action from another, you can invoke the action method explicitly. This code will make `index` set both `@budget` and `@list`:

```
class ListController < ApplicationController
  def index
    shopping_list and render :action => 'shopping_list'
    @budget = 87.50
  end
end
```

Another consequence of this "envelope" behavior is that you must never call `render` twice within a single client request (the same goes for `render`'s cousin `redirect_to`, which also seals a message in an envelope).

If you write code like the following, Rails will complain. You're giving it two sealed envelopes, and it doesn't know which to open:

```
class ListController < ApplicationController
  def plain_and_fancy
    render :action => 'plain_list'
    render :action => 'fancy_list'
  end
end
```

But the following code is fine, because any given request will only trigger one branch of the `if`/`else` clause. Whatever happens, `render` will only be called once per request.

```
class ListController < ApplicationController
  def plain_or_fancy
    if params[:fancy]
      render :action => 'fancy_list'
    else
      render :action => 'plain_list'
    end
  end
end
```

With `redirect_to`, if you want to force your action method to stop running, you can put a `return` statement immediately after your call to `render`. This code does not set the `@budget` variable, because execution never gets past the `return` statement:

```
class ListController < ApplicationController
  def index
    render :action => 'shopping_list' and return
    @budget = 87.50         # This line won't be run.
  end
end
```

See Also

- Recipe 15.4, "Redirecting to a Different Location"
- Recipe 15.14, "Refactoring the View into Partial Snippets of Views," shows examples of calling `render` within a view template

15.6 Integrating a Database with Your Rails Application

Problem

You want your web application to store persistent data in a relational database.

Solution

The hardest part is setting things up: creating your database and hooking Rails up to it. Once that's done, database access is as simple as writing Ruby code.

To tell Rails how to access your database, open your application's `config/database.yml` file. Assuming your Rails application is called `mywebapp`, it should look something like this:

```
development:
adapter: mysql
database: mywebapp_development
host: localhost
username: root
password:

test:
adapter: mysql
database: mywebapp_test
host: localhost
username: root
password:

production:
adapter: mysql
database: mywebapp
host: localhost
username: root
password:
```

For now, just make sure the `development` section contains a valid username and password, and that it mentions the correct adapter name for your type of database (see Chapter 13 for the list).

Now create a database table. As with so much else, Rails does a lot of the database work automatically if you follow its conventions. You can override the conventions if necessary, but for now it's easiest to go along with them.

The name of the table must be a pluralized noun: for instance, "people", "tasks", "items".

The table must contain an auto-incrementing primary key field called `id`.

For this example, use a database tool or a `CREATE DATABASE` SQL command to create a `mywebapp_development` database (see the chapter introduction for Chapter 13 if you need help doing this). Then create a table in that database called `people`. Here's the SQL to create a `people` table in MySQL; you can adapt it for your database.

```
use mywebapp_development;

DROP TABLE IF EXISTS 'people';
CREATE TABLE 'people' (
'id' INT(11) NOT NULL AUTO_INCREMENT,
'name' VARCHAR(255),
'email' VARCHAR(255),
PRIMARY KEY (id)
) ENGINE=InnoDB;
```

Now go to the command line, change into the web application's root directory, and type `./script/generate model Person`. This generates a Ruby class that knows how to manipulate the `people` table.

```
$ ./script/generate model Person
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/person.rb
create test/unit/person_test.rb
create test/fixtures/people.yml
```

Notice that your model is named `Person`, even though the table was named `people`. If you abide by its conventions, Rails automatically handles these pluralizations for you (see Recipe 15.7 for details).

Your web application now has access to the `people` table, via the `Person` class. Again from the command line, run this command:

```
$ ./script/runner 'Person.create(:name => "John Doe", \
:email => "john@doe.com")'
```

That code creates a new entry in the `people` table. (If you've read Recipe 13.11, you'll recognize this as ActiveRecord code.)

To access this person from your application, create a new controller and a view to go along with it:

```
$ ./script/generate controller people list
exists app/controllers/
```

```
exists app/helpers/
create app/views/people
exists test/functional/
create app/controllers/people_controller.rb
create test/functional/people_controller_test.rb
create app/helpers/people_helper.rb
create app/views/people/list.rhtml
```

Edit *app/view/people/list.rhtml* so it looks like this:

```
<!-- list.rhtml -->
<ul>
<% Person.find(:all).each do |person| %>
<li>Name: <%= person.name %>, Email: <%= person.email %
></li>
<% end %>
</ul>
```

Start the Rails server, visit *http://localhost:3000/people/list/*, and you'll see John Doe listed.

The `Person` model class is accessible from all parts of your Rails application: your controllers, views, helpers, and mailers.

Discussion

Up until now, the applications created in these recipes have been using only controllers and views.* The `Person` class, and its underlying database table, give us for the first time the Model portion of the Model-View-Controller triangle.

A relational database is usually the best place to store real-world models, but it's difficult to program a relational database directly. Rails uses the ActiveRecord library to hide the `people` table behind a `Person` class. Methods like `Person.find` let you search the `person` database table without writing SQL; the results are automatically converted into `Person` objects. The basics of ActiveRecord are covered in Recipe 13.11.

The `Person`.find method takes a lot of optional arguments. If you pass it an integer, it will look for the `person` entry whose unique ID is that integer, and return an appropriate `Person` object. The `:all` and `:first` symbols grab all entries from the table (an array of `Person` objects), or only the first person that matches. You can limit or order your dataset by specifying `:limit` or `:order`; you can even set raw SQL conditions via `:conditions`.

Here's how to find the first five entries in the `people` table that have email addresses. The result will be a list containing five `Person` objects, ordered by their `name` fields.

```
Person.find(:all,
     :limit => 5,
```

* More precisely, our models have been embedded in our controllers, as ad hoc data structures like hardcoded shopping lists.

```
        :order => 'name',
        :conditions => 'email IS NOT NULL')
```

The three different sections of `config/database.yml` specify the three different databases used at different times by your Rails application:

Development database
: The database you use when working on the application. Generally filled with test data.

Test database
: A scratch database used by the unit testing framework when running tests for your application. Its data is populated automatically by the unit testing framework.

Production database
: The database mode to use when your web site is running with live data.

Unless you explicitly setup Rails to run in production or test mode, it defaults to development mode. So to get started, you only need to make sure the `development` portion of `database.yml` is set up correctly.

See Also

- Chapter 13
- Recipe 13.11, "Using Object Relational Mapping with ActiveRecord"
- Recipe 13.13, "Building Queries Programmatically"
- Recipe 13.14, "Validating Data with ActiveRecord"
- ActiveRecord can't do everything that SQL can. For complex database operations, you'll need to use `DBI` or one of the Ruby bindings to specific kinds of database; these topics too are covered in Recipe 13.15, "Preventing SQL Injection Attacks," which gives more on the format of the `database.yml` file

15.7 Understanding Pluralization Rules

Problem

You want to understand and customize Rails's rules for automatically pluralizing nouns.

Solution

You can use Rails' pluralization functionality in any part of your application, but ActiveRecord is the only major part of Rails that does pluralization automatically. ActiveRecord generally expects table names to be pluralized noun phrases and the corresponding model classes to be singular versions of the same noun phrases.

So when you create a model class, you should always use a singular name. Rails automatically pluralizes:

- The corresponding table name for the model
- `has_many` relations
- `has_and_belongs_to_many` relations

For example, if you create a `LineItem` model, the table name automatically becomes `line_items`. Note also that the table name has been lowercased, and the word break indicated by the original camelcase is now conveyed with an underscore.

If you then create an `Order` model, the corresponding table needs to be called `orders`. If you want to describe an order that has many line items, the code would look like this:

```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

Like the name of the table it references, the symbol used in the `has_many` relation is pluralized and underscored. The same goes for the other relationships between tables, like `has_and_belongs_to_many`.

Discussion

ActiveRecord pluralizes these names to make your code read more like an English sentence: `has_many :line_items` can be read "has many line items". If pluralization confuses you, you can disable it by setting `ActiveRecord::Base.pluralize_table_names` to false. In Rails, the simplest way to do this is to put the following code in `config/environment.rb`:

```
Rails::Initializer.run do |config|
  config.active_record.pluralize_table_names = false
end
```

If your application knows specific words that ActiveRecord does not know how to pluralize, you can define your own pluralization rules by manipulating the `Inflector` class. Let's say that the plural of "foo" is "fooze", and you've build an application to manage fooze. In Rails, you can specify this transformation by putting the following code in `config/environment.rb`:

```
Inflector.inflections do |inflect|
  inflect.plural /^(foo)$/i, '\1ze'
  inflect.singular /^(foo)ze/i, '\1'
end
```

In this case, it's simpler to use the `irregular` method:

```
Inflector.inflections do |inflect|
  inflect.irregular 'foo', 'fooze'
end
```

If you have nouns that should never be inflected (usually because they are mass nouns, or because their plural form is the same as their singular form), you can pass them into the uncountable method:

```
Inflector.inflections do |inflect|
  inflect.uncountable ['status', 'furniture', 'fish', 'sheep']
end
```

The Inflector class is part of the activesupport gem, and you can use it outside of ActiveRecord or Rails as a general way of pluralizing English words. Here's a standalone Ruby program:

```
require 'rubygems'
require 'active_support/core_ext'

'blob'.pluralize                                # => "blobs"
'child'.pluralize                               # => "children"
'octopus'.pluralize                             # => "octopi"
'octopi'.singularize                            # => "octopus"
'people'.singularize                            # => "person"

'goose'.pluralize                               # => "gooses"
Inflector.inflections { |i| i.irregular 'goose', 'geese' }
'goose'.pluralize                               # => "geese"

'moose'.pluralize                               # => "mooses"
Inflector.inflections { |i| i.uncountable 'moose' }
'moose'.pluralize                               # => "moose"
```

See Also

- Recipe 13.11, "Using Object Relational Mapping with ActiveRecord"

15.8 Creating a Login System

Problem

You want your application to support a login system based on user accounts. Users will log in with a unique username and password, as in most commercial and community web sites.

Solution

Create a users table that contains nonnull username and password fields. The SQL to create this table should look something like this MySQL example:

```
use mywebapp_development;
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(255) NOT NULL,
```

```
  `password` VARCHAR(40) NOT NULL,
  PRIMARY KEY (`id`)
);
```

Enter the main directory of the application and generate a `User` model corresponding to this table:

```
$ ./script/generate model User
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/user.rb
      create  test/unit/user_test.rb
      create  test/fixtures/users.yml
```

Open the generated file *app/models/user.rb* and edit it to look like this:

```
class User < ActiveRecord::Base
  validates_uniqueness_of :username
  validates_confirmation_of :password, :on => :create
  validates_length_of :password, :within => 5..40

  # If a user matching the credentials is found, returns the User object.
  # If no matching user is found, returns nil.
  def self.authenticate(user_info)
    find_by_username_and_password(user_info[:username],
                                  user_info[:password])
  end
end
```

Now you've got a `User` class that represents a user account, and a way of validating a username and password against the one stored in the database.

Discussion

The simple `User` model given in the Solution defines a method for doing username/password validation, and some validation rules that impose limitations on the data to be stored in the `users` table. These validation rules tell `User` to:

- Ensure that each username is unique. No two users can have the same username.
- Ensure that, whenever the `password` attribute is being set, the `password_confirmation` attribute has the same value.
- Ensure that the value of the `password` attribute is between 5 and 40 characters long.

Now let's create a controller for this model. It'll have a `login` action to display the login page, a `process_login` action to check the username and password, and a `logout` action to deauthenticate a logged-in session. So that the user accounts will actually do something, we'll also add a `my_account` action:

```
$ ./script/generate controller user login process_login logout my_account
      exists  app/controllers/
      exists  app/helpers/
```

```
      create  app/views/user
      exists  test/functional/
      create  app/controllers/user_controller.rb
      create  test/functional/user_controller_test.rb
      create  app/helpers/user_helper.rb
      create  app/views/user/login.rhtml
      create  app/views/user/process_login.rhtml
      create  app/views/user/logout.rhtml
```

Edit *app/controllers/user_controller.rb* to define the three actions:

```
class UserController < ApplicationController
  def login
    @user = User.new
    @user.username = params[:username]
  end

  def process_login
    if user = User.authenticate(params[:user])
      session[:id] = user.id # Remember the user's id during this session
      redirect_to session[:return_to] || '/'
    else
      flash[:error] = 'Invalid login.'
      redirect_to :action => 'login', :username => params[:user][:username]
    end
  end

  def logout
    reset_session
    flash[:message] = 'Logged out.'
    redirect_to :action => 'login'
  end

  def my_account
  end
end
```

Now for the views. The `process_login` and `logout` actions just redirect to other actions, so we only need views for `login` and `my_account`. Here's a view for `login`:

```
<!-- app/views/user/login.rhtml -->
<% if @flash[:message] %><div><%= @flash[:message] %></div><% end %>
<% if @flash[:error] %><div><%= @flash[:error] %></div><% end %>

<%= form_tag :action => 'process_login' %>
 Username: <%= text_field "user", "username" %>&#x00A;
  Password: <%= password_field "user", "password" %>&#x00A;
  <%= submit_tag %>
<%= end_form_tag %>
```

The `@flash` instance variable is a hashlike object used to store temporary messages for the user between actions. When the `logout` action sets `flash[:message]` and redirects to `login`, or `process_login` sets `flash[:error]` and redirects to `login`, the results are available to the view of the `login` action. Then they get cleared out.

Here's a very simple view for `my_account`:

```
<!-- app/views/user/my_account.rhtml -->
<h1>Account Info</h1>

<p>Your username is <%= User.find(session[:id]).username %>
```

Create an entry in the `users` table, start the server, and you'll find that you can log in from *http://localhost:3000/user/login*, and view your account information from *http://localhost:3000/user/my_account*.

```
$ ./script/runner 'User.create(:username => "johndoe", \
                               :password => "changeme")'
```

There's just one missing piece: you can visit the `my_account` action even if you're not logged in. We don't have a way to close off an action to unauthenticated users. Add the following code to your *app/controllers/application.rb* file:

```
class ApplicationController < ActionController::Base
  before_filter :set_user

protected
  def set_user
    @user = User.find(session[:id]) if @user.nil? && session[:id]
  end

  def login_required
    return true if @user
    access_denied
    return false
  end

  def access_denied
    session[:return_to] = request.request_uri
    flash[:error] = 'Oops. You need to login before you can view that page.'
    redirect_to :controller => 'user', :action => 'login'
  end
end
```

This code defines two filters, `set_user` and `login_required`, which you can apply to actions or controllers. The `set_user` filter is run on every action (because we pass it into `before_filter` in `ApplicationController`, the superclass of all our controllers). The `set_user` method sets the instance variable `@user` if the user is logged in. Now information about the logged-in user (if any) is available throughout your application. Action methods and views can use this instance variable like any other. This is useful even for actions that don't require login: for instance, your main layout view might display the name of the logged-in user (if any) on every page.

You can prohibit unauthenticated users from using a specific action or controller by passing the symbol for the `login_required` method into `before_filter`. Here's how to protect the `my_account` action defined in *app/controllers/user_controller.rb*:

```
class UserController < ApplicationController
  before_filter :login_required, :only => :my_account
end
```

Now if you try to use the `my_account` action without being logged in, you'll be redirected to the login page.

See Also

- Recipe 13.14, "Validating Data with ActiveRecord"
- Recipe 15.6, "Integrating a Database with Your Rails Application"
- Recipe 15.9, "Storing Hashed User Passwords in the Database"
- Recipe 15.11, "Setting and Retrieving Session Information"
- Rather than doing this work yourself, you can install the `login_generator` gem and use its `login` generator: it will give your application a `User` model and a controller that implements a password-based authentication system; see *http://wiki.rubyonrails.com/rails/pages/LoginGenerator*; also see *http://wiki.rubyonrails.com/rails/pages/AvailableGenerators* for other generators (including the more sophisticated `model_security_generator`)

15.9 Storing Hashed User Passwords in the Database

Problem

The database table defined in Recipe 15.8 stores users' passwords as plain text. This is a bad idea: if someone compromises the database, she will have all of your users' passwords. It's best to store a secure hash of the password instead. That way, you don't have the password (so no one can steal it), but you can verify that a user knows his password.

Solution

Recreate the `users` table from Recipe 15.8 so that instead of a `password` field, it has a `hashed_password field`. Here's some MySQL code to do that:

```
use mywebapp_development;
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(255) NOT NULL,
  `hashed_password` VARCHAR(40) NOT NULL,
  PRIMARY KEY (id)
);
```

Open the file *app/models/user.rb* created in Recipe 15.8, and edit it to look like this:

```
require 'sha1'

class User < ActiveRecord::Base
  attr_accessor :password
  attr_protected :hashed_password
  validates_uniqueness_of :username
```

```
  validates_confirmation_of :password,
    :if => lambda { |user| user.new_record? or not user.password.blank? }
  validates_length_of :password, :within => 5..40,
    :if => lambda { |user| user.new_record? or not user.password.blank? }

  def self.hashed(str)
    SHA1.new(str).to_s
  end

  # If a user matching the credentials is found, returns the User object.
  # If no matching user is found, returns nil.
  def self.authenticate(user_info)
    user = find_by_username(user_info[:username])
    if user && user.hashed_password == hashed(user_info[:password])
      return user
    end
  end

  private
  before_save :update_password

  # Updates the hashed_password if a plain password was provided.
  def update_password
    if not password.blank?
      self.hashed_password = self.class.hashed(password)
    end
  end
end
```

Once you do this, your application will work as before (though you'll have to convert any preexisting user accounts to the new password format). You don't need to modify any of the controller or view code, because the `User.authenticate` method works the same way it did before. This is one of the benefits of separating business logic from presentation logic.

Discussion

There are now three pieces to our user model. The first is the enhanced validation code. The user model now:

- Provides getters and setters for the `password` attribute.
- Makes sure that the `hashed_password` field in the database can't be accessed from the outside.
- Ensures that each user has a unique username.

When a new user is created, or when the password is changed, `User` ensures:

- That the value of the `password_confirmation` attribute is equal to the value of the `password` attribute.
- That the password is between 5 and 40 characters long.

The second section of code defines `User` class methods as before. We add one new class-level method, `hashed`, which performs the hashing function on a plaintext password. If we want to change hashing mechanisms in the future, we only have to change this method (and migrate any existing passwords).

The third piece of code in the model is a private instance method, `update_password`, which synchronizes the plaintext `password` attribute with the hashed version in the database. The call to `before_save` sets up this method to be called before a `User` object is saved to the database. This way you can change a user's password by setting `password` to its plaintext value, instead of doing the hash yourself.

See Also

- Recipe 13.14, "Validating Data with ActiveRecord"
- Recipe 15.8, "Creating a Login System"

15.10 Escaping HTML and JavaScript for Display

Problem

You want to display data that might contain HTML or JavaScript without making browsers render it as HTML or interpret the JavaScript. This is especially important when displaying data entered by users.

Solution

Pass a string of data into the `h( )` helper function to escape its HTML entities. That is, instead of this:

```
<%= @data %>
```

Write this:

```
<%=h @data %>
```

The `h( )` helper function converts the following characters into their HTML entity equivalents: ampersand (`&`), double quote (`"`), left angle bracket (`<`), and right angle bracket (`>`).

Discussion

You won't find the definition for the `h( )` helper function anywhere in the Rails source code, because it's a shortcut for ERb's built-in helper function `html_escape( )`.

JavaScript is deployed within HTML tags like `<SCRIPT>`, so escaping an HTML string will neutralize any JavaScript in the HTML. However, sometimes you need to escape just the JavaScript in a string. Rails adds a helper function called `escape_javascript( )` that you can use. This function doesn't do much: it just turns line breaks into the

string "\n", and adds backslashes before single and double quotes. This is handy when you want to use arbitrary data in your own JavaScript code:

```
<!-- index.rhtml -->
<script lang="javascript">
var text = "<%= escape_javascript @javascript_alert_text %>";
alert(text);
</script>
```

See Also

- Chapter 11

15.11 Setting and Retrieving Session Information

Problem

You want to associate some data with each distinct web client that's using your application. The data needs to persist across HTTP requests.

Solution

You can use cookies (see Recipe 15.12) but it's usually simpler to put the data in a user's session. Every visitor to your Rails site is automatically given a session cookie. Rails keys the value of the cookie to a hash of arbitrary data on the server.

Throughout your entire Rails application, in controllers, views, helpers, and mailers, you can access this hash by calling a method called `session`. The objects stored in this hash are persisted across requests by the same web browser.

This code in a controller tracks the time of a client's first visit to your web site:

```
class IndexController < ApplicationController
  def index
    session[:first_time] ||= Time.now
  end
end
```

Within your view, you can write the following code to display the time:*

```
<!-- index.rhtml -->
You first visited this site on <%= session[:first_time] %>.

That was <%= time_ago_in_words session[:first_time] %> ago.
```

* The helper function `time_ago_in_words( )` calculates how long it's been since a certain time and returns English text such as "about a minute" or "5 hours" or "2 days". This is a nice, easy way to give the user a perspective on what a date means.

Discussion

Cookies and sessions are very similar. They both store persistent data about a visitor to your site. They both let you implement stateful operations on top of HTTP, which has no state of its own. The main difference between cookies and sessions is that with cookies, all the data is stored on your visitors' computers in little cookie files. With sessions, all the data is stored on the web server. The client only keeps a small session cookie, which contains a unique ID that's tied to the data on the server. No personal data is ever stored on the visitor's computer.

There are a number of reasons why you might want to use sessions instead of cookies:

- A cookie can only store four kilobytes of data.
- A cookie can only store a string value.
- If you store personal information in a cookie, it can be intercepted unless all of a client's requests are encrypted with SSL. Even then, cross-site scripting attacks may be able to read the client cookie and retrieve the sensitive information.

On the other hand, cookies are useful when:

- The information is not sensitive and not very large.
- You don't want to store session information about each visitor on your server.
- You need speed from your application, and not every page needs to access the session data.

Generally, it's a better idea to use sessions than to store data in cookies.

You can include model objects in your session: this can save a lot of trouble over retrieving the same objects from the database on every request. However, if you are going to do this, it's a good idea to list in your application controller all the models you'll be putting into the session. This reduces the risk that Rails won't be able to deserialize the objects when retrieving them from the session store.

```
class ApplicationController < ActionController::Base
  model :user, :ticket, :item, :history
end
```

Then you can put ActiveRecord objects into a session:

```
class IndexController < ApplicationController
  def index
    session[:user] ||= User.find(params[:id])
  end
end
```

If your site doesn't need to store any information in sessions, you can disable the feature by adding the following code to your `app/controllers/application.rb` file:

```
class ApplicationController < ActionController::Base
  session :off
end
```

As you may have guessed, you can also use the `session` method to turn sessions off for a single controller:

```
class MyController < ApplicationController
  session :off
end
```

You can even bring it down to an action level:

```
class MyController < ApplicationController
  session :off, :only => ['index']

  def index
    #This action will not have any sessions available to it
  end
end
```

The session interface is intended for data that persists over many actions, possibly over the user's entire visit to the site. If you just need to pass an object (like a status message) to the next action, it's simpler to use the `flash` construct described in Recipe 15.8:

```
flash[:error] = 'Invalid login.'
```

By default, Rails sessions are stored on the server via the `PStore` mechanism. This mechanism uses `Marshal` to serialize session data to temporary files. This approach works well for small sites, but if your site will be getting a lot of visitors or you need to run your Rails application concurrently on multiple servers, you should explore some of the alternatives.

The three main alternatives are `ActiveRecordStore`, `DRbStore`, and `MemCacheStore`. `ActiveRecordStore` keeps session information in a database table: you can set up the table by running `rake create_sessions_table` on the command line. Both `DRbStore` and `MemCacheStore` create an in-memory hash that's accessible over the network, but they use different libraries.

Ruby comes with a standard library called `DRb` that allows you to share objects (including hashes) over the network. Ruby also has a binding to the Memcached daemon, which has been used to help scale web sites like Slashdot and LiveJournal. Memcached works like a direct store into RAM, and can be distributed automatically over various computers without any special configuration.

To change the session storing mechanism, edit your *config/environment.rb* file like this:

```
Rails::Initializer.run do |config|
  config.action_controller.session_store = :active_record_store
end
```

See Also

- Recipe 15.8, "Creating a Login System," has an example using `flash`
- Recipe 15.12, "Setting and Retrieving Cookies"
- Recipe 16.10, "Sharing a Hash Between Any Number of Computers"

- Recipe 16.16, "Storing Data on Distributed RAM with MemCached"
- *http://wiki.rubyonrails.com/rails/pages/HowtoChangeSessionOptions*

15.12 Setting and Retrieving Cookies

Problem

You want to set a cookie from within Rails.

Solution

Recall from Recipe 15.11 that all Rails controllers, views, helpers, and mailers have access to a method called `sessions` that returns a hash of the current client's session information. Your controllers, helpers, and mailers (but not your views) also have access to a method called `cookies`, which returns a hash of the current client's HTTP cookies.

To set a cookie for a user, simply set a key/value pair in that hash. For example, to keep track of how many pages a visitor has looked at, you might set a "visits" cookie:

```
class ApplicationController < ActionController::Base
  before_filter :count_visits

  private

  def count_visits
    value = (cookies[:visits] || '0').to_i
    cookies[:visits] = (value + 1).to_s
    @visits = cookies[:visits]
  end
end
```

The call to `before_filter` tells Rails to run this method before calling any action method. The `private` declaration makes sure that Rails doesn't think the `count_visits` method is itself an action method that the public can view.

Since cookies are not directly available to views, `count_visits` makes the value of the `:visits` cookie available as the instance variable `@visits`. This variable can be accessed from a view:

```
<!-- index.rhtml -->
You've visited this website's pages <%= @visits %> time(s).
```

HTTP cookie values can only be strings. Rails can automatically convert some values to strings, but it's safest to store only string values in `cookies`. If you need to store objects that can't easily be converted to and from strings, you should probably store them in the `session` hash instead.

Discussion

There may be times when you want more control over your cookies. For instance, Rails cookies expire by default when the user closes their browser session. If you want to change the browser expiration time, you can give `cookies` a hash that contains an `:expires` key and a time to expire the cookie. The following cookie will expire after one hour:*

```
cookies[:user_id] = { :value => '123', :expires => Time.now + 1.hour}
```

Here are some other options for a cookie hash passed into `cookies`.

The domain to which this cookie applies:

```
:domain
```

The URL path to which this cookie applies (by default, the cookie applies to the entire domain: this means that if you host multiple applications on the same domain, their cookies may conflict):

```
:path
```

Whether this cookie is secure (secure cookies are only transmitted over HTTPS connections; the default is false):

```
:secure
```

Finally, Rails provides a quick and easy way to delete cookies:

```
cookies.delete :user_id
```

Of course, every Ruby hash implements a `delete` method, but the `cookies` hash is a little different. It includes special code so that not only does calling `delete` remove a key-value pair from the `cookies` hash, it removes the corresponding cookie from the user's browser.

See Also

- Recipe 3.5, "Doing Date Arithmetic"
- Recipe 15.11, "Setting and Retrieving Session Information," has a discussion of when to use `cookies` and when to use `session`

* Rails extends Ruby's numeric classes to include some very helpful methods (like the `hour` method shown here). These methods convert the given unit to seconds. For example, `Time.now + 1.hour` is the same as `Time.now + 3600`, since `1.hour` returns the number of seconds in an hour. Other helpful methods include `minutes`, `hours`, `days`, `months`, `weeks`, and `years`. Since they all convert to numbers of seconds, you can even add them together like `1.week + 3.days`.

15.13 Extracting Code into Helper Functions

Problem

Your views are getting cluttered with Ruby code.

Solution

Let's create a controller with a fairly complex view to see how this can happen:

```
$ ./scripts/generate controller list index
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/list
      exists  test/functional/
      create  app/controllers/list_controller.rb
      create  test/functional/list_controller_test.rb
      create  app/helpers/list_helper.rb
      create  app/views/list/index.rhtml
```

Edit `app/controllers/list_controller.rb` to look like this:

```
class ListController < ApplicationController
  def index
    @list = [1, "string", :symbol, ['list']]
  end
end
```

Edit `app/views/list/index.rhtml` to contain the following code. It iterates over each element in `@list`, and prints out its index and the SHA1 hash of its object ID:

```
<!-- app/views/list/index.rhtml -->
<ul>
<% @list.each_with_index do |item, i| %>
  <li class="<%= i%2==0 ? 'even' : 'odd' %>"><%= i %>:
    <%= SHA1.new(item.id.to_s) %></li>
<% end %>
</ul>
```

This is pretty messy, but if you've done much web programming it should also look sadly familiar.

To clean up this code, we're going to move some of it into the *helper* for the controller. In this case, the controller is called `list`, so its helper lives in `app/helpers/list_helper.rb`.

Let's create a helper function called `create_li`. Given an object and its position in the list, this function creates an `<LI>` tag suitable for use in the `index` view:

```
module ListHelper
  def create_li(item, i)
    %{<li class="#{ i%2==0 ? 'even' : 'odd' }">#{i}:
      #{SHA1.new(item.id.to_s)}</li>}
  end
end
```

The `list` controller's views have access to all the functions defined in `ListHelper`. We can clean up the `index` view like so:

```
<!-- app/views/list/index.rhtml -->
<ul>
<% @list.each_with_index do |item, i| %>
  <%= create_li(item, i) %>
<% end %>
</ul>
```

Your helper functions can do anything you can normally do from within a view, so they are a great way to abstract out the heavy lifting.

Discussion

The purpose of helper functions is to create more maintainable code, and to enforce a good division of labor between the programmers and the UI designers. Maintainable code is easier for the programmers to work on, and when it's in helper functions it's out of the way of the designers, who can tweak the HTML here and there without having to sifting through code.

A good rule of thumb for when to use helpers is to read the code aloud. If it sounds like nonsense to someone familiar with HTML, or it makes up more than a short English sentence, hide it in a helper.

The flip side of this is that you should minimize the amount of HTML generated from within the helpers. That way the UI designers, or other people familiar with HTML, won't wander your code, wondering where to find the bit of HTML that needs tweaking.

Although helper functions are useful and used very often, Rails also provides *partials*, another way of extracting code into smaller chunks.

See Also

- Recipe 15.14, "Refactoring the View into Partial Snippets of Views," has more on partials

15.14 Refactoring the View into Partial Snippets of Views

Problem

Your view doesn't contain a lot of Ruby code, but it's still becoming more complicated than you'd like. You'd like to refactor the view logic into separate, reusable templates.

Solution

You can refactor a view template into multiple templates called partials. One template can include another by calling the `render` method, first seen in Recipe 15.5.

Let's start with a more complex version of the view shown in Recipe 15.5:

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
<% @list.each do |item| %>
 <li><%= item.name %> -
  <%= link_to 'Delete', {:action => 'delete', :id => item.id},
              :post => true %>
 </li>
<% end %>
</ul>

<h2>Add a new item</h2>

<%= form_tag :action => 'new' %>
 Item: <%= text_field "product", "name" %>&#xOOA;
 <%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

Here's the corresponding controller class, and a dummy `ListItem` class to serve as the model:

```
# app/controllers/list_controller.rb
class ListController < ActionController::Base
  def shopping_list
    @list = [ListItem.new(4, 'aspirin'), ListItem.new(199, 'succotash')]
  end

  # Other actions go here: add, delete, etc.
  # ...
end

class ListItem
  def initialize(id, name)
    @id, @name = id, name
  end
end
```

The view has two parts: the first part lists all the items, and the second part prints a form to add a new item. An obvious first step is to split out the new item form.

We can do this by creating a partial view to print the new item form. To do this, create a new file within *app/views/list/* called *_new_item_form.rhtml*. The underscore in front of the filename indicates that it is a partial view, not a full-fledged view for an action called `new_item_form`. Here's the partial file.

```
<!-- app/views/list/_new_item_form.rhtml -->

<%= form_tag :action => 'new' %>
Item: <%= text_field "item", "value" %>&#x00A;
<%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

To include a partial, call the `render` method from within a template. Here is the `_new_item_form` partial integrated into the main view. The view looks exactly the same, but the code is better organized.

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
<% @list.each do |item| %>
 <li><%= item.name %> -
  <%= link_to 'Delete', {:action => 'delete', :id => item.id},
              :post => true %>
 </li>
<% end %>
</ul>
<%= render :partial => 'new_item_form' %>
```

Even though the filename starts with an underscore, when you call the partial, you omit the underscore.

Discussion

Partial views inherit all the instance variables provided by the controller, so they have access to the same instance variables as the parent view. That's why we didn't have to change any of the form code for the `_new_item_form` partial.

We can create a second partial to factor out the code that prints the <LI> tag for each list item. Here's *_list_item.rhtml*:

```
<!-- app/views/list/_list_item.rhtml -->
<li><%= list_item.name %> -
<%= link_to 'Delete', {:action => 'delete', :id => list_item.id},
            :post => true %>
</li>
```

And here's the revised main view:

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
<% @list.each do |item| %>
  <%= render :partial => 'list_item', :locals => {:list_item => item} %>
<% end %>
</ul>

<%= render :partial => 'new_item_form' %>
```

Partial views do *not* inherit local variables from their parent view, so the `item` variable needs to be passed in to the partial, in a special hash called `:locals`. It's accessible in the partial as `list_item`, because that's the name it was given in the hash.

This scenario, iterating over an `Enumerable` and rendering a partial for each element, is very common in web applications, so Rails provides a shortcut. We can simplify our main view even more by passing our array into `render` (as the `:collection` parameter) and having it do the iteration for us:

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
 <%= render :collection => @list, :partial => 'list_item' %>
</ul>

<%= render :partial => 'new_item_form' %>
```

The partial is rendered once for every element in `@list`. Each list element is made available as the local variable `list_item`. In case you haven't guessed, this name comes from the name of the partial itself: `render` automatically gives `_foo.rhtml` a local variable called `foo`.

`list_item_counter` is another variable that is set automatically (again, the name mirrors the name of the template). `list_item_counter` is the current item's index in the collection undergoing iteration. This variable can be handy if you want alternating list items to show up in different styles:

```
<!-- app/views/list/_list_item.rhtml -->
<li><%= list_item.name %> -
<% css_class = list_item_counter % 2 == 0 ? 'a' : 'b' %>
<%= link_to 'Delete', {:action => 'delete', :id => list_item.id},
            {'class' => css_class}, :post => true %>
</li>
```

When there's no collection present, you can pass a single object into a partial by specifying an `:object` argument to `render`. This is simpler than creating a whole hash of `:locals` just to pass one object. As with `:collection`, the object will be made available as a local variable whose name is based on the name of the partial.

Here's an example: we'll send the shopping list into the `new_item_form.rhtml` partial, so that the new item form can print a more verbose message. Here's the change to `shopping_list.rhtml`:

```
<%= render :partial => 'new_item_form', :object => @list %>
```

Here's the new version of `_new_item_form.rhtml`:

```
<!-- app/views/list/_new_item_form.rhtml -->
<h2>Add a new item to the <%= new_item_form.size %> already in this
list</h2>
```

```
<%= form_tag :action => 'new' %>
 Item: <%= text_field "product", "name" %>
 <%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

See Also

- Recipe 15.5, "Displaying Templates with Render"

15.15 Adding DHTML Effects with script.aculo.us

Problem

You want to add fancy effects such as fades to your application, without writing any JavaScript.

Solution

Every Rails application comes bundled with some JavaScript libraries that allow you to create Ajax and DHTML effects. You don't even have to write JavaScript to enable DHTML in your Rails web site.

First edit your main layout template (see Recipe 15.3) to call `javascript_include_tag` within your `<HEAD>` tag:

```
<!-- app/views/layouts/application.rhtml -->

<html>
  <head>
    <title>My Web App</title>
    <%= javascript_include_tag "prototype", "effects" %>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

Now within your views you can call the `visual_effect` method to accomplish the DHTML tricks found in the `script.aculo.us` library.

Here's an example of the "highlight" effect:

```
<p id="important">Here is some important text, it will be highlighted
when the page loads.</p>

<script type="text/javascript">
<%= visual_effect(:highlight, "important", :duration => 1.5) %>
</script>
```

Here's an example of the "fade" effect:

```
<p id="deleted">Here is some old text, it will fade away when the page
loads.</p>
```

```
<script type="text/javascript">
<%= visual_effect(:fade, "deleted", :duration => 1.0) %>
</script>
```

Discussion

The sample code snippets above are triggered when the page loads, because they're enclosed in <SCRIPT> tags. In a real application, you'll probably display text effects in response to user actions: deleted items might fade away, or the selection of one item might highlight related items. Here's an image that gets squished when you click the link below it:

```
<img id="to-squish" src="bug.jpg">
<%=link_to_function("Squish the bug!", visual_effect(:squish, "to-squish"))%>
```

The JavaScript code generated by the `visual_effect` method looks a lot like the arguments you passed into the method. For instance, this piece of a Rails view:

```
<script type="text/javascript">
<%= visual_effect(:fade, 'deleted-text', :duration => 1.0) %>
</script>
```

Generates this JavaScript:

```
<script type="text/javascript">
new Effect.Fade("deleted-text", {duration:1.0});
</script>
```

This similarity means that documentation for the `script.aculo.us` library is almost directly applicable to `visual_effect`. It also means that if you feel more comfortable writing straight JavaScript, your code will still be fairly understandable to someone who knows `visual_effect`.

The following table lists many of the effects available in Rails 1.0.

JavaScript initialization	Rails initialization
new Effect.Highlight	visual_effect(:highlight)
new Effect.Appear	visual_effect(:appear)
new Effect.Fade	visual_effect(:fade)
new Effect.Puff	visual_effect(:puff)
new Effect.BlindDown	visual_effect(:blind_down)
new Effect.BlindUp	visual_effect(:blind_up)
new Effect.SwitchOff	visual_effect(:switch_off)
new Effect.SlideDown	visual_effect(:slide_down)
new Effect.SlideUp	visual_effect(:slide_up)
new Effect.DropOut	visual_effect(:drop_out)
new Effect.Shake	visual_effect(:shake)
new Effect.Pulsate	visual_effect(:pulsate)

JavaScript initialization	Rails initialization
new Effect.Squish	visual_effect(:squish)
new Effect.Fold	visual_effect(:fold)
new Effect.Grow	visual_effect(:grow)
new Effect.Shrink	visual_effect(:shrink)
new Effect.ScrollTo	visual_effect(:scroll_to)

See Also

- The `script.aculo.us` demo (*http://wiki.script.aculo.us/scriptaculous/show/CombinationEffectsDemo*)
- Recipe 15.3, "Creating a Layout for Your Header and Footer," has more on layout templates
- Recipe 15.17, "Creating an Ajax Form"

15.16 Generating Forms for Manipulating Model Objects

Problem

You want to define actions that let a user create or edit objects stored in the database.

Solution

Let's create a simple model, and then build forms for it. Here's some MySQL code to create a table of key-value pairs:

```
use mywebapp_development;
DROP TABLE IF EXISTS items;
CREATE TABLE `items` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(255) NOT NULL default '',
  `value` varchar(40) NOT NULL default '[empty]',
  PRIMARY KEY (`id`)
);
```

Now, from the command line, create the model class, along with a controller and views:

```
$ ./script/generate model Item
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/item.rb
      create  test/unit/item_test.rb
      create  test/fixtures/items.yml
      create  db/migrate
      create  db/migrate/001_create_items.rb
```

```
$ ./script/generate controller items new create edit
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/items
      exists  test/functional/
      create  app/controllers/items_controller.rb
      create  test/functional/items_controller_test.rb
      create  app/helpers/items_helper.rb
      create  app/views/items/new.rhtml
      create  app/views/items/edit.rhtml
```

The first step is to customize a view. Let's start with *app/views/items/new.rhtml*. Edit it to look like this:

```
<!-- app/views/items/new.rhtml -->

<%= form_tag :action => "create" %>
 Name: <%= text_field "item", "name" %><br />
 Value: <%= text_field "item", "value" %><br />
 <%= submit_tag %>
<%= end_form_tag %>
```

All these method calls generate HTML: `form_tag` opens a `<FORM>` tag, `submit_tag` generates a submit button, and so on. You can type out the same HTML by hand and Rails won't care, but it's easier to make method calls, and it makes your templates neater.

The `text_field` call is a little more involved. It creates an `<INPUT>` tag that shows up in the HTML form as a text entry field. But it also binds the value of that field to one of the members of the `@item` instance variable. This code creates a text entry field that's bound to the `name` member of `@item`:

```
<%= text_field "item", "name" %>
```

But what's the `@item` instance variable? Well, it's not defined yet, because we're still using the generated controller. If you try to access the page `/items/new` page right now, you may get an error complaining about an unexpected `nil` value. The `nil` value is the `@item` variable, which gets used (in `text_field` calls) without ever being defined.

Let's customize the `ItemsController` class so that the `new` action sets the `@item` instance variable properly. We'll also implement the `create` action so that something actually happens when the user hits the submit button on our generated form.

```
class ItemsController < ApplicationController
  def new
    @item = Item.new
  end

  def create
    @item = Item.create(params[:item])
    redirect_to :action => 'edit', :id => @item.id
  end
end
```

Now if you access the /items/new page, you'll see what you'd expect: a form with two text entry fields. The "Name" field will be blank, and the "Value" field will contain the default database value of "[empty]".

Fill out the form and submit, and a new row will be created in the items table. You'll be redirected to the edit action, which doesn't exist yet. Let's create it now. Here's the controller part (note the similarity between ItemsController#edit and ItemsController#create above):

```
class ItemsController < ApplicationController
  def edit
    @item = Item.find(params[:id])

    if request.post?
      @item.update_attributes(params[:item])
      redirect_to :action => 'edit', :id => @item.id
    end
  end
end
```

In fact, the edit action is so similar to the create action that its form can be almost identical. The only differences are in the arguments to form_tag:

```
<!-- app/views/items/edit.rhtml -->

<%= form_tag :action => "edit", :id => @item.id %>
 Name: <%= text_field "item", "name" %><br />
 Value: <%= text_field "item", "value" %><br />
 <%= submit_tag %>
<%= end_form_tag %>
```

Discussion

This is probably the most common day-to-day task faced by web developers. It's so common that Rails comes with a tool called scaffold that generates this kind of code for you. If you'd invoked generate this way instead of with the arguments given above, Rails would have generate code for the actions given in the Solution, plus a few more:

```
$ ./script/generate scaffold Items
```

Starting off with scaffolding doesn't mean you can get away with not knowing how Rails form generation works, because you'll definitely want to customize the scaffolding code.

There are two places in our code where magic happens. The first is the text_field call in the view, which is explained in the Solution. It binds a member of an object (@item.name, for instance) to an HTML form control. If you view the source of the /items/new page, you will see that the form fields look something like this:

```
Name: <input type="text" name="item[name]" value="" /><br />
Value <input type="text" name="item[value]" value="[empty]" /><br />
```

These special field names are used by the second piece of magic, located in the calls to `Item.create` (in `new`) and `Item#update_attributes`. In both cases, an `Item` object is fed a hash of new values for its members. This hash is embedded into the `params` hash, which contains CGI form values.

The names of the HTML form fields (`item[name]` and `item[value]`) translate into a `params` hash that looks like this:

```
{
  :item => {
    :name => "Name of the item",
    :value => "Value of the item"
  },
  :controller => "items",
  :action => "create"
}
```

So this line of code:

```
Item.create(params[:item])
```

is effectively the same as this line:

```
Item.create(:name => "Name of the item", :value => "Value of the item")
```

The call to `Item#update_attributes` in the `edit` action works exactly the same way.

As mentioned above, the views for `edit` and `new` are very similar, differing only in the destination of the form. With some minor refactoring, we can remove one of the view files completely.

A call to `<%= form_tag %>` without any parameters at all sets the form destination to the current URL. Let's change the *new.rhtml* file appropriately:

```
<!-- app/views/items/new.rhtml -->
<%= form_tag %>

Name: <%= text_field "item", "name" %>&#xOOA;
Value: <%= text_field "item", "value" %>&#xOOA;

<%= submit_tag %>
<%= end_form_tag %>
```

Now the `new.rhtml` view is suitable for use by both `new` and `edit`. We just need to change the `new` action to call the `create` method (since the form doesn't go there anymore), and change the `edit` action to render `new.rhtml` instead of `edit.rhtml` (which can be removed):

```
class ItemsController < ApplicationController
  def new
    @item = Item.new
    create if request.post?
  end
```

```
  def edit
    @item = Item.find(params[:id])

    if request.post?
      @item.update_attributes(params[:item])
      redirect_to :action => 'edit', :id => @item.id and return
    end
    render :action => 'new'
  end
end
```

Remember from Recipe 15.5 that a `render` call only specifies the template file to be used. The `render` call in `edit` won't actually call the `new` method, so we don't need to worry about the `new` method overwriting our value of `@item`.

In real life, there would be enough differences in the content surrounding the `add` and `edit` forms to a separate view for each action. However, there's usually enough similarity between the forms themselves that they can be refactored into a single partial view (see Recipe 15.14) which both views share. This is a great example of the DRY (Don't Repeat Yourself) principle. If there is a single form for both the `add` and `edit` views, it's easier and less error-prone to maintain that form as the database schema changes.

See Also

- Recipe 15.5, "Displaying Templates with Render"
- Recipe 15.14, "Refactoring the View into Partial Snippets of Views"

15.17 Creating an Ajax Form

Problem

You want to build a web application that's responsive and easy to use. You don't want your users to spend lots of time waiting around for the browser to redraw the screen.

Solution

You can use JavaScript to make the browser's `XMLHTTPRequest` object send data to the server, without dragging the user through the familiar (but slow) page refresh. This technique is called Ajax,* and Rails makes it easy to use Ajax without writing or knowing any JavaScript.

* This doesn't quite stand for Asynchronous JavaScript and XML. The origins of the term Ajax are now a part of computing mythology, but it is not an acronym.

Before you can do Ajax in your web application, you must edit your application's main layout template so that it calls the `javascript_include_tag` method within its `<HEAD>` tag. This is the same change made in Recipe 15.15:

```
<!-- app/views/layouts/application.rhtml -->

<html>
  <head>
    <title>My Web App</title>
    <%= javascript_include_tag "prototype", "effects" %>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

Let's change the application from Recipe 15.16 so that the `new` action is AJAX-enabled (if you followed that recipe all the way through, and made the `edit` action use `new.rhtml` instead of `edit.rhtml`, you'll need to undo that change and make `edit` use its own view template).

We'll start with the view template. Edit *app/views/items/new.rhtml* to look like this:

```
<!-- app/views/items/new.rhtml -->
<div id="show_item"></div>

        <%= form_remote_tag :url => { :action => :create },
        :update => "show_item",
        :complete => visual_effect(:highlight, "show_item") %>

 Name: <%= text_field "item", "name" %><br />
 Value: <%= text_field "item", "value" %><br />
 <%= submit_tag %>
<%= end_form_tag %>
```

Those small changes make a standard HTML form into an Ajax form. The main difference is that we call `form_remote_tag` instead of `form_tag`. The other differences are the arguments we pass into that method.

The first change is that we put the `:action` parameter inside a hash passed into the `:url` option. Ajax forms have more options associated with them than a normal form, so you can't describe its form action as simply as you can with `form_tag`.

When the user clicks the submit button, the form values are serialized and sent to the destination action (in this case, `create`) in the background. The `create` action processes the form submission as before, and returns a snippet of HTML.

What happens to this HTML? That's what the `:update` option is for. It tells Rails to take the result of the form submission, and stick it into the element with the HTML ID of "show_item". This is why we added that `<div id="show_item">` tag to the top of the template: that's where the response from the server goes.

The last change to the new.rhtml view is the :complete option. This is a callback argument: it lets you specify a string of JavaScript code that will be run once an Ajax request is complete. We use it to highlight the response from the server once it shows up.

That's the view. We also need to modify the create action in the controller so that when you make an Ajax form submission, the server returns a snippet of HTML. This is the snippet that's inserted into the "show_item" element on the browser side. If you make a regular (nonAjax) form submission, the server can behave as it does in Recipe 15.16, and send an HTTP redirect.* Here's what the controller class needs to look like:

```
class ItemsController < ApplicationController
  def new
    @item = Item.new
  end

  def create
    @item = Item.create(params[:item])
      if request.xml_http_request?
      render :action => 'show', :layout => false
    else
      redirect_to :action => 'edit', :id => @item.id
    end
  end

  def edit
    @item = Item.find(params[:id])

    if request.post?
       @item.update_attributes(params[:item])
       redirect_to :action => 'edit', :id => @item.id
    end
  end
end
```

This code references a new view, show. It's the tiny HTML snippet that's returned by the server, and stuck into the "show_element" tag by the web browser. We need to define it:

```
<!-- app/views/items/show.rhtml -->

Your most recently created item:<br />
Name: <%= @item.name %><br />
Value: <%= @item.value %><br />
<hr>
```

Now when you use *http://localhost:3000/items/new* to add new items to the database, you won't be redirected to the edit action. You'll stay on the new page, and the

* This will happen if someone's using your application with JavaScript turned off.

results of your form submission will be displayed above the form. This makes it easy to create many new items at once.

Discussion

Recipe 15.16 shows how to submit data to a form in the traditional way: the user clicks a "submit" button, the browser sends a request to the server, the server returns a response page, and the browser renders the response page.

Recently, sites like Gmail and Google Maps have popularized techniques for sending and receiving data without a page refresh. Collectively, these techniques are called Ajax. Ajax is a very useful tool for improving your application's response time and usability.

An Ajax request is a real HTTP request to one of your application's actions, and you can deal with it as you would any other request. Most of the time, though, you won't be returning a full HTML page. You'll just be returning a snippet of data. The web browser will be sending the Ajax request in the context of a full web page (which you served up earlier) that knows how to handle the response snippet.

You can define JavaScript callbacks at several points throughout the lifecycle of an Ajax request. One callback, `:complete`, was used above to highlight the snippet after inserting it into the page. This table lists the other callbacks.

Callback name	Callback description
:loading	Called when the web browser begins to load the remote document.
:loaded	Called when the browser has finished loading the remote document.
:interactive	Called when the user can interact with the remote document, even if it has not finished loading.
:success	Called when the XMLHttpRequest is completed, and the HTTP status code is in the 2XX range.
:failure	Called when the XMLHttpRequest is completed, and the HTTP status code is not in the 2XX range.
:complete	Called when the XMLHttpRequest is complete. If `:success` and/or `:failure` are also present, runs after they do.

15.18 Exposing Web Services on Your Web Site

Problem

You want to offer SOAP and XML-RPC web services from your web application.

Solution

Rails comes with a built-in web service generator that makes it easy to expose a controller's actions as web services. You don't have to spend time writing WSDL files or even really know how SOAP and XML-RPC work.

Here's a simple example. First, follow the directions in Recipe 15.16 to create a database table named `items`, and to generate a model for that table. Don't generate a controller.

Now, run this from the command line:

```
./script/generate web_service Item add edit fetch
      create  app/apis/
      exists  app/controllers/
      exists  test/functional/
      create  app/apis/item_api.rb
      create  app/controllers/item_controller.rb
      create  test/functional/item_api_test.rb
```

This creates an `item` controller that supports three actions: `add`, `edit`, and `fetch`. But instead of web application actions with `.rhtml` views, these are web service actions that you access with SOAP or XML-RPC.

A Ruby method doesn't care about the data types of the objects it accepts as arguments, or the data type of its return value. But a SOAP or XML-RPC web service method does care. To expose a Ruby method through a SOAP or XML-RPC interface, we need to define type information for its signature. Open up the file *app/apis/item_api.rb* and edit it to look like this:

```
class ItemApi < ActionWebService::API::Base
  api_method :add, :expects => [:string, :string], :returns => [:int]
  api_method :edit, :expects => [:int, :string, :string], :returns => [:bool]
  api_method :fetch, :expects => [:int], :returns => [Item]
end
```

Now we need to implement the actual web service interface. Open *app/controllers/item_controller.rb* and edit it to look like this:

```
class ItemController < ApplicationController
  wsdl_service_name 'Item'

  def add(name, value)
    Item.create(:name => name, :value => value).id
  end

  def edit(id, name, value)
    Item.find(id).update_attributes(:name => name, :value => value)
  end

  def fetch(id)
    Item.find(id)
  end
end
```

Discussion

The `item` controller now implements SOAP and XML-RPC web services for the `items` table. This controller can live alongside an `items` controller that implements a traditional web interface.*

The URL to the XML-RPC API is *http://www.yourserver.com/item/api*, and the URL to the SOAP API is *http://www.yourserver.com/item/service.wsdl*. To test these services, here's a short Ruby script that calls the web service methods through a SOAP client:

```
require 'soap/wsdlDriver'

wsdl = "http://localhost:3000/item/service.wsdl"
item_server = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver

item_id = item_server.add('foo', 'bar')

if item_server.edit(item_id, 'John', 'Doe')
  puts 'Hey, it worked!'
else
  puts 'Back to the drawing board...'
end
# Hey, it worked!

item = item_server.fetch(item_id)
item.class                                          # => SOAP::Mapping::Object
item.name                                           # => "John"
item.value                                          # => "Doe"
```

Here's the XML-RPC equivalent:

```
require 'xmlrpc/client'
item_server = XMLRPC::Client.new2('http://localhost:3000/item/api')

item_id = item_server.call('Add', 'foo', "bar")
if item_server.call('Edit', item_id, 'John', 'Doe')
  puts 'Hey, it worked!'
else
  puts 'Back to the drawing board...'
end
# Hey, it worked!

item = item_server.call('Fetch', item_id)
# => {"name"=>"John", "id"=>2, "value"=>"Doe"}
item.class                                          # => Hash
```

* You can even add your web interface actions to the `ItemController` class. Then a single controller will implement both the traditional web interface and the web service interface. But you can't define a web application action with the same name as a web service action, because a controller class can contain only one method with a given name.

See Also

- Matt Biddulph's article "REST on Rails" describes how to create REST-style web services on top of Rails (*http://www.xml.com/pub/a/2005/11/02/rest-on-rails.html*)
- Recipe 16.3, "Writing an XML-RPC Client," and Recipe 16.4, "Writing a SOAP Client"
- Recipe 16.5, "Writing a SOAP Server," shows a nonRails implementation of SOAP web services

15.19 Sending Mail with Rails

Problem

You want to send an email from within your Rails application: perhaps a confirmation of an order, or notification that some action has been taken on a user's behalf.

Solution

The first is to generate some mailer infrastructure. Go to the application's base directory and type this command:

```
./script/generate mailer Notification welcome
      exists  app/models/
      create  app/views/notification
      exists  test/unit/
      create  test/fixtures/notification
      create  app/models/notification.rb
      create  test/unit/notification_test.rb
      create  app/views/notification/welcome.rhtml
      create  test/fixtures/notification/welcome
```

We're giving the name "Notification" to the mailing center of the application; it's somewhat analogous to a controller in the web interface. The mailer is set up to generate a single email, called "welcome": this is analagous to an action with a view template.

Now open `app/models/notification.rb` and edit it to look like this:

```
class Notification < ActionMailer::Base
  def welcome(user, sent_at=Time.now)
    @subject =  'A Friendly Welcome'
    @recipients = user.email
    @from =  'admin@mysite.com'
    @sent_on =    sent_at
    @body = {
      :user => user,
      :sent_on => sent_at
    }
```

```
    attachment 'text/plain' do |a|
      a.body = File.read('rules.txt')
    end
  end
end
```

The subject of the email is "A Friendly Welcome", and it's sent to the user's email address from the address "*admin@mysite.com*". It's got an attachment taken from the disk file `rules.txt` (relative to the root directory of your Rails application).

Although the file `notification.rb` is within the `models/` directory, it acts like a controller in that each of its email messages has an associated view template. The view for the welcome email is in `app/views/notification/welcome.rhtml`, and it acts almost the same as the view of a normal controller.

The most important difference is that mailer views do not have access to the instance variables of the mailer. To set instance variables for mailers, you pass a hash of those variables to the `body` method. The keys become instance variable names and the values become their values. In `notification.rb`, we make two instance variables available to the welcome view, `@user` and `@sent_on`. Here's the view itself:

```
<!-- app/views/notification/welcome.rhtml -->

Hello, <%= @user.name %>, and thanks for signing up at <%= @sent_on
%>. Please print out the attached set of rules and keep them in a
prominent place; they help keep our community running smoothly. Be
sure to pay special attention to sections II.4 ("Assignment of
Intellectual Property Rights") and XIV.21.a ("Dispute Resolution
Through Ritual Combat").
```

To send the welcome email from your Rails application, add the following code to either a controller, a model, or an observer:

```
Notification.deliver_welcome(user)
```

Here, the `user` variable can be any object that implements `#name` and `#email`, the two methods called in the `welcome` method and in the template.

Discussion

You never call the `Notification#welcome` method directly. In fact, `Notification#welcome` is not even available, since it's an instance method, and you never instantiate a `Notification` object directly. The `ActionMailer::Base` class defines a `method_missing` implementation that looks at all calls to undefined class methods. This is why you call `deliver_welcome` even though you never defined it.

The `welcome.rhtml` template given above generates plaintext email. To send HTML emails, simply add the following code to `Notification#welcome`:

```
content_type 'text/html'
```

Now your templates can generate HTML; email clients will recognize the format of the email and render it appropriately.

Sometimes you'll want more control over the delivery process—for example, when you're unit-testing your ActionMailer classes. Instead of calling `deliver_welcome` to send out an email, you can call `create_welcome` to get the email as a Ruby object. These "create" methods return `TMail` objects, which you can examine or manipulate as necessary.

If your local web server is incapable of sending email, you can modify `environment.rb` to contact a remote SMTP server:

```
Rails::Initializer.run do |config|
  config.action_mailer.server_settings = {
    :address => 'someserver.com',
    :user_name => 'uname',
    :password => 'passwd',
    :authentication => 'cram_md5'
  }
end
```

See Also

- Recipe 10.8, "Responding to Calls to Undefined Methods"
- Recipe 14.5, "Sending Mail," has more on ActionMailer and SMTP settings

15.20 Automatically Sending Error Messages to Your Email

Problem

You want to receive a descriptive email message every time one of your users encounters an application error.

Solution

Any errors that occur while running your application are sent to the `ActionController::Base#log_error` method. If you've set up a mailer (as shown in Recipe 15.19) you can override this method and have it send mail to you. Your code should look something like this:

```
class ApplicationController < ActionController::Base

private
   def log_error(exception)
      super
      Notification.deliver_error_message(exception,
        clean_backtrace(exception),
        session.instance_variable_get("@data"),
```

```
        params,
        request.env
      )
    end
end
```

That code rounds up a wide variety of information about the state of the Rails request at the time of the failure. It captures the exception object, the corresponding backtrace, the session data, the CGI request parameters, and the values of all environment variables.

The overridden `log_error` calls `Notification.deliver_error_messsage`, which assumes you've created a mailer called "Notification", and defined the method `Notification.error_message`. Here's the implementation:

```
class Notification < ActionMailer::Base
  def error_message(exception, trace, session, params, env, sent_on = Time.now)
    @recipients         = 'me@mydomain.com'
    @from               = 'error@mydomain.com'
    @subject            = "Error message: #{env['REQUEST_URI']}"
    @sent_on            = sent_on
    @body = {
      :exception => exception,
      :trace => trace,
      :session => session,
      :params => params,
      :env => env
    }
  end
end
```

The template for this email looks like this:

```
<!-- app/views/notification/error_message.rhtml -->

Time: <%= Time.now %>
Message: <%= @exception.message %>
Location: <%= @env['REQUEST_URI'] %>
Action: <%= @params.delete('action') %></td></tr>
Controller: <%= @params.delete('controller') %></td></tr>
Query: <%= @env['QUERY_STRING'] %></td></tr>
Method: <%= @env['REQUEST_METHOD'] %></td></tr>
SSL: <%= @env['SERVER_PORT'].to_i == 443 ? "true" : "false" %>
Agent: <%= @env['HTTP_USER_AGENT'] %>

Backtrace
<%= @trace.to_a.join("</p>\n<p>") %>

Params
<% @params.each do |key, val| -%>
* <%= key %>: <%= val.to_yaml %>
<% end -%>
```

```
Session
<% @session.each do |key, val| -%>
* <%= key %>: <%= val.to_yaml %>
<% end -%>

Environment
<% @env.each do |key, val| -%>
* <%= key %>: <%= val %>
<% end -%>
```

Discussion

`ActionController::Base#log_error` gives you the flexibility to handle errors however you like. This is especially useful if your Rails application is hosted on a machine to which you have limited access: you can have errors sent to you, instead of written to a file you might not be able to see. Or you might prefer to record the errors in a database, so that you can look for patterns.

The method `ApplicationController#log_error` is declared private to avoid confusion. If it weren't private, all of the controllers would think they had a `log_error` action defined. Users would be able to visit `/<controller>/log_error` and get Rails to act strangely.

See Also

- Recipe 15.19, "Sending Mail with Rails"

15.21 Documenting Your Web Site

Problem

You want to document the controllers, models, and helpers of your web application so that the developers responsible for maintaining the application can understand how it works.

Solution

As with any other Ruby program, you document a Rails application by adding specially-formatted commands to your code. Here's how to add documentation to the `FooController` class and one of its methods:

```
# The FooController controller contains miscellaneous functionality
# rejected from other controllers.
class FooController < ApplicationController
  # The set_random action sets the @random_number instance variable
  # to a random number.
  def set_random
    @random_number = rand*rand
  end
end
```

The documentation for classes and methods goes before their declaration, not after.

When you've finished adding documentation comments to your application, go to your Rails application's root directory and issue the `rake appdoc` command:

```
$ rake appdoc
```

This Rake task runs RDoc for your Rails application and generates a directory called `doc/app`. This directory contains a web site with the aggregate of all your documentation comments, cross-referenced against the source code. Open the `doc/app/index.rhtml` file in any web browser, and you can browse the generated documentation.

Discussion

Your RDoc comments can contain markup and special directives: you can describe your arguments in definition lists, and hide a class or method from documentation with the `:nodoc:` directive. This is covered in Recipe 17.11.

The only difference between Rails applications and other Ruby programs is that Rails comes with a Rakefile that defines an `appdoc` task. You don't have to find or write one yourself.

You probably already put inline comments *inside* your methods, describing the action as it happens. Since the RDoc documentation contains a formatted version of the original source code, these comments will be visible to people going through the RDoc. These comments are formatted as Ruby source code, though, not as RDoc markup.

See Also

- Recipe 17.11, "Documenting Your Application"
- Chapter 19, especially Recipe 19.2, "Automatically Generating Documentation"
- The RDoc for RDoc (*http://rdoc.sourceforge.net/doc/index.html*)

15.22 Unit Testing Your Web Site

Problem

You want to create a suite of automated tests that test the functionality of your Rails application.

Solution

Rails can't write your test code any more than it can write your views and controllers for you, but it does make it easy to organize and run your automated tests.

When you use the `./script/generate` command to create controllers and models, not only do you save time, but you also get a generated framework for unit and functional tests. You can get pretty good test coverage by filling in the framework with tests for the functionality you write.

So far, all the examples in this chapter have run against a Rails application's development database, so you only needed to make sure that the `development` section of your *config/database.yml* file was set up correctly. Unit test code runs on your application's test database, so now you need to set up your `test` section as well. Your `mywebapp_test` database doesn't have to have any tables in it, but it must exist and be accessible to Rails.

When you generate a model with the `generate` script, Rails also generates a unit test script for the model in the *test* directory. It also creates a *fixture*, a YAML file containing test data to be loaded into the `mywebapp_test` database. This is the data against which your unit tests will run:

```
./script/generate model User
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/user.rb
      create  test/unit/user_test.rb
      create  test/fixtures/users.yml
      create  db/migrate
      create  db/migrate/001_create_users.rb
```

When you generate a controller with `generate`, Rails creates a functional test script for the controller:

```
./script/generate users list
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/users
      exists  test/functional/
      create  app/controllers/users_controller.rb
      create test/functional/users_controller_test.rb
      create  app/helpers/users_helper.rb
      create  app/views/users/list.rhtml
```

As you write code in the model and controller classes, you'll write corresponding tests in these files.

To run the unit and functional tests, invoke the `rake` command in your home directory. The default Rake task runs all of your tests. If you run it immediately after generating your test files, it'll look something like this:

```
$ rake
(in /home/lucas/mywebapp)
/usr/bin/ruby1.8 "test/unit/user_test.rb"
Started
.
Finished in 0.048702 seconds.
```

```
1 tests, 1 assertions, 0 failures, 0 errors
/usr/bin/ruby1.8 "test/functional/users_controller_test.rb"
Started
.
Finished in 0.024615 seconds.

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

Discussion

All the lessons for writing unit tests in other languages and in other Ruby programs (see Recipe 17.7) apply to Rails. Rails does some accounting for you, and it defines some useful new assertions (see below), but you still have to do the work. The rewards are the same, too: you can modify and refactor your code with confidence, knowing that if something breaks, your tests will break. You'll hear about the problem immediately and you'll be able to fix it more quickly.

Let's see what Rails has generated for us. Here's a generated *test/unit/user_test.rb*:

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

class UserTest < Test::Unit::TestCase
  fixtures :users

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

A good start, but `test_truth` is kind of tautological. Here's a slightly more realistic test:

```
class UserTest
  def test_first
    assert_kind_of User, users(:first)
  end
end
```

This code fetches the first element from the `users` table, and asserts that ActiveRecord turns it into a `User` object. This isn't testing our `User` code (we haven't written any) so much as it's testing Rails and ActiveRecord, but it shows you the kind of assertion that makes for good unit tests.

But how does `users(:first)` return anything? The test suite runs against the `mywebapp_test` database, and we didn't even put any tables in it, much less sample data.

We didn't, but Rails did. When you run the test suite, Rails copies the schema of the development database to the test database. Instead of running every test against whatever data happens to exist in the development database, Rails loads special test data from YAML files called fixtures. The fixture files contain whatever database data you need to test: objects that only exist to be deleted by a test, strange relationships between rows in different tables, or anything else you need.

In the example above, the fixture for the `users` table was loaded by the line `fixtures :users`. Here's the generated fixture for the `User` model, in *test/fixtures/users.yml*:

```
first:
  id: 1
another:
  id: 2
```

Before running the unit tests, Rails reads this file, creates two rows in the `users` table, and defines aliases for them (`:first` and `:another`) so you can refer to them in your unit tests. It then defines the `users` method (like so much else, this method name is based on the name of the model). In `test_first`, the call to `users(:first)` retrieves the `User` object corresponding to `:first` in the fixture: the object with ID 1.

Here's another unit test:

```
class UserTest
  def test_another
    assert_kind_of User, users(:another)
    assert_equal 2, users(:another).id
    assert_not_equal users(:first), users(:another)
  end
end
```

Rails adds the following Rails-specific assertions to Ruby's `Test::Unit`:

- `assert_dom_equal`
- `assert_dom_not_equal`
- `assert_generates`
- `assert_no_tag`
- `assert_recognizes`
- `assert_redirected_to`
- `assert_response`
- `assert_routing`
- `assert_tag`
- `assert_template`
- `assert_valid`

See Also

- "Testing the Rails" is a guide to unit and functional testing in Rails (*http://manuals.rubyonrails.com/read/book/5*)
- Rails 1.1 supports integration testing as well, for testing the interactions between controllers and actions; see *http://rubyonrails.com/rails/classes/ActionController/IntegrationTest.html* and *http://jamis.jamisbuck.org/articles/2006/03/09/integration-testing-in-rails-1-1*

- The ZenTest library inclues Test::Rails, which lets you write separate tests for your views and controllers (*http://rubyforge.org/projects/zentest/*)
- Read about fixtures at *http://ar.rubyonrails.org/classes/Fixtures.html*
- Read about the assertions that Rails adds to `Test::Unit` at *http://rails.rubyonrails.com/classes/Test/Unit/Assertions.html*
- Recipe 15.6, "Integrating a Database with Your Rails Application"
- Recipe 17.7, "Writing Unit Tests"
- Chapter 19

15.23 Using breakpoint in Your Web Application

Problem

Your Rails application has a bug that you can't find using log messages. You need a heavy-duty debugging tool that lets you inspect the full state of your application at any given point.

Solution

The `breakpoint` library lets you stop the flow of code and drop into `irb`, an interactive Ruby session. Within `irb` you can inspect the variables local to the current scope, modify those variables, and resume execution of the normal flow of code. If you have ever spent hours trying to track down a bug by placing logging messages everywhere, you'll find that `breakpoint` gives you a much easier and more straightforward way to debug.

But how can you run an interactive console program from a web application? The answer is to have a console program running beforehand, listening for calls from the Rails server.

The first step is to run `./script/breakpointer` on the command line. This command starts a server that listens over the network for breakpoint calls from the Rails server. Keep this program running in a terminal window: this is where the `irb` session will start up:

```
$ ./script/breakpointer
No connection to breakpoint service at druby://localhost:42531
Tries to connect will be made every 2 seconds...
```

To trigger an `irb` session, you can call the `breakpoint` method anywhere you like from your Rails application—within a model, controller, or helper method. When execution reaches that point, processing of the incoming client request will stop, and an `irb` session will start in your terminal. When you quit the session, processing of the request will resume.

Discussion

Here's an example. Let's say you've written the following controller, and you're having trouble modifying the name attribute of an `Item` object.

```
class ItemsController < ApplicationController
  def update
    @item = Item.find(params[:id])
    @item.value = '[default]'
    @item.name = params[:name]
    @item.save
    render :text => 'Saved'
  end
end
```

You can put a `breakpoint` call in the `Item` class, like this:

```
class Item < ActiveRecord::Base
  attr_accessor :name, :value

  def name=(name)
    super
    breakpoint
  end
end
```

Accessing the URL *http://localhost:3000/items/update/123?name=Foo* calls `ItemsController#update`, which finds `Item` number 123 and then calls its `name=` method. The call to `name=` triggers the breakpoint. Instead of rendering the text "Saved", the site seems to hang and become unresponsive to requests.

But if you return to the terminal running the `breakpointer` server, you'll see that an interactive Ruby session has started. This session allows you to play with all the local variables and methods at the point where the breakpoint was called:

```
Executing break point "Item#name=" at item.rb:4 in `name='
  irb:001:0> local_variables
  => ["name", "value", "_", "__"]
  irb:002:0> [name, value]
  => ["Foo", "[default]"]
 irb:003:0> [@name, @value]
  => ["Foo", "[default]"]
  irb:004:0> self
  => #<Item:0x292fbe8 @name="Foo", @value="[default]">
  irb:005:0> self.value = "Bar"
  => "Bar"
  irb:006:0> save
 => true
  irb:006:0> exit

Server exited. Closing connection...
```

Once you finish, type `exit` to terminate the interactive Ruby session. The Rails application continues running at the place it left off, rendering "Saved" as expected.

By default, breakpoints are named for the method in which they appear. You can pass a string into `breakpoint` to get a more descriptive name. This is especially helpful if one method contains several breakpoints:

```
breakpoint "Trying to set Item#name, just called super"
```

Instead of calling `breakpoint` directly, you can also call `assert`, a method which takes a code block. If the block evaluates to `false`, Ruby calls `breakpoint`; otherwise, things continue as normal. Using `assert` lets you set breakpoints that are only called when something goes wrong (called "conditional breakpoints" in traditional debuggers):

```
1.upto 10 do |i|
  assert { Person.find(i) }
  p = Person.find(i)
  p.update_attribute(:name, 'Lucas')
end
```

If all of the required `Person` objects are found, the breakpoint is never called, because `Person.find` always returns true. If one of the `Person` objects is missing, Ruby calls the `breakpoint` method and you get an `irb` session to investigate.

Breakpoint is a powerful tool that can vastly simplify your debugging process. It can be hard to understand the true power of it until you try it yourself, so go through the solution with your own code to toy around with it.

See Also

- Recipe 17.10, "Using breakpoint to Inspect and Change the State of Your Application," covers `breakpoint` in more detail.
- *http://wiki.rubyonrails.com/rails/show/HowtoDebugWithBreakpoint*