O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
http://www.onlamp.com/pub/a/onlamp/2005/03/03/rails.html
See this if you're having trouble printing code examples

Rolling with Ruby on Rails, Part 2

by Curt Hibbs
03/03/2005

Welcome back!

In Rolling with Ruby on Rails, I barely scratched the surface of what you can do with Ruby on Rails. I didn't talk about data validation or database transactions, and I did not mention callbacks, unit testing, or caching. There was hardly a mention of the many helpers that Rails includes to make your life easier. I can't really do justice to all of these topics in the space of this article, but I will go into details on some of them and present a brief overview of the rest, with links to more detailed information.

Also, I purposely did not go into any detail about the Ruby programming language. If you are interested in a brief treatment of the whys behind the Ruby and Rails code that you saw in Part 1, then I highly recommend reading Amy Hoy's blog entry Really Getting Started in Rails.

Before I cover this stuff, I want to complete the homework items from the end of Part 1:

- There is no longer any way to delete a recipe. Add a delete button (or link) to the edit template.
- On the main recipes page, there aren't any links for the pages that let you manipulate categories. Fix that.
- It would be nice to have a way to display only those recipes in a particular category. For example, maybe I'd like to see a list of all snack recipes, or all beverage recipes. On the page that lists all recipes, make each category name a link to a page that will display all of the recipes in that category.

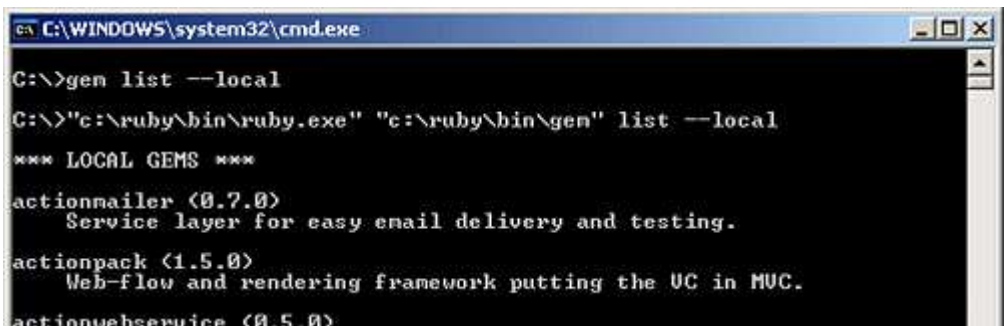
One alert reader pointed out that after adding categories, it was no longer possible to add any new recipes because the *new recipe* action (as provided by the scaffolding) had no way to assign a category, and this caused the *list recipes* action to produce an error. This also needs fixing.

Are you ready? Let's begin!

Updating Ruby on Rails

When I wrote Part 1, the current version of Rails was 0.9.3. At the time of this writing, Rails is up to version 0.10.0 and has some useful new features. I will use Rails 0.10.0 for this article. If you installed Rails after February 24, 2005, you already have 0.10.0 installed.

Figure 1 shows how to see what RubyGems you have installed (and their version numbers). As with Part 1, I am working on a Windows system, so you will need to translate if you use a different platform.



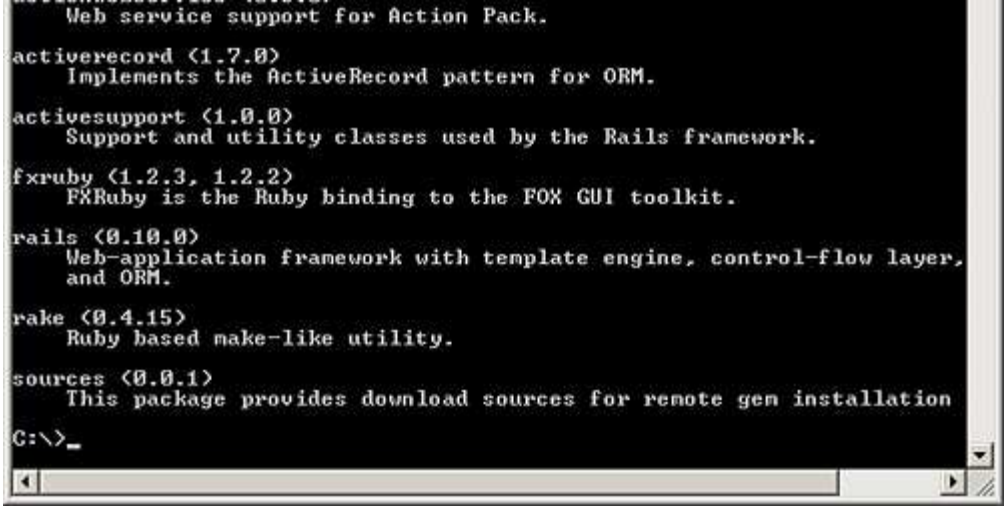


Figure 1. Listing installed RubyGems

Open a command window and run the command:

```
gem list --local
```

Tip: the command `gem list --remote` will show all the available RubyGems on the remote gem server on rubyforge.org.

If you don't have Rails 0.10.0 (or later) installed, then you will need to rerun the command:

```
gem install rails
```

MySQL security update

In Part 1, I recommended that you leave the MySQL root password blank because (at the time of writing) Rails did not support MySQL's new password protocol. Many of you were not happy with this state of affairs, and to make matters worse, there is now a virus that exploits password vulnerabilities in MySQL on Windows.

Happily, starting with version 0.9.4, Rails now supports the new password protocol.

New scaffold feature

Rails has a new scaffold feature, which I won't explore here, but it's cool enough that I want to make sure you know about it. This is best illustrated by an example.

Part 1 showed how to create a recipe model and controller with the commands:

```
ruby script\generate model Recipe
ruby script\generate controller Recipe
```

I then instantiated the scaffolding by inserting `scaffold :recipe` into the `RecipeController` class. The resulting CRUD controllers and view templates were created on the fly and are not visible for inspection.

The technique described above still works, but you now have another option. Run the command:

```
ruby script\generate scaffold Recipe
```

This generates both the model and the controller, plus it creates scaffold code and view templates for all CRUD operations. This allows you to see the scaffold code and modify it to meet your needs. Be careful using this if you've already created models, controllers, or view templates, as it will overwrite any existing files as it creates the scaffold code.

Completing the Recipe Application

It's time to round out the recipe application a bit. After that I'll present some other features of Rails that I'm sure you'll want to know about.

Remember that I created my cookbook application in the directory *c:\rails\cookbook*; all paths used in this article assume this base directory. If you chose a different location, please be sure to make the proper adjustments to the application paths you see in this article.

If you completed Part 1 using anything earlier than Rails 0.10.0, you will need to update your cookbook application to work with Rails 0.10.0. You can either follow the upgrading instructions or download my cookbook source code, which has the upgrades already.

For those of you who are cheating (you know who you are) and plan to just download my source code without going through Part 1, you will also need to create a database named `cookbook` in MySQL and populate it using cookbook.sql.

Creating a new recipe with a category

Because the code still relies on the scaffolding to create new recipes, there is no way to assign a category to a recipe. This wouldn't be so bad--except that the page created to list all recipes assumes that every recipe will have a category, and it generates an error if this is not true. That means that in the way I left things in Part 1, if you add a new recipe, you'll receive errors while trying to list them.

The fix is to take over the new action from the scaffolding just as I showed already with the edit action. Edit *c:\rails\cookbook\app\controllers\recipe_controller.rb* and add a `new` method like in Figure 2.

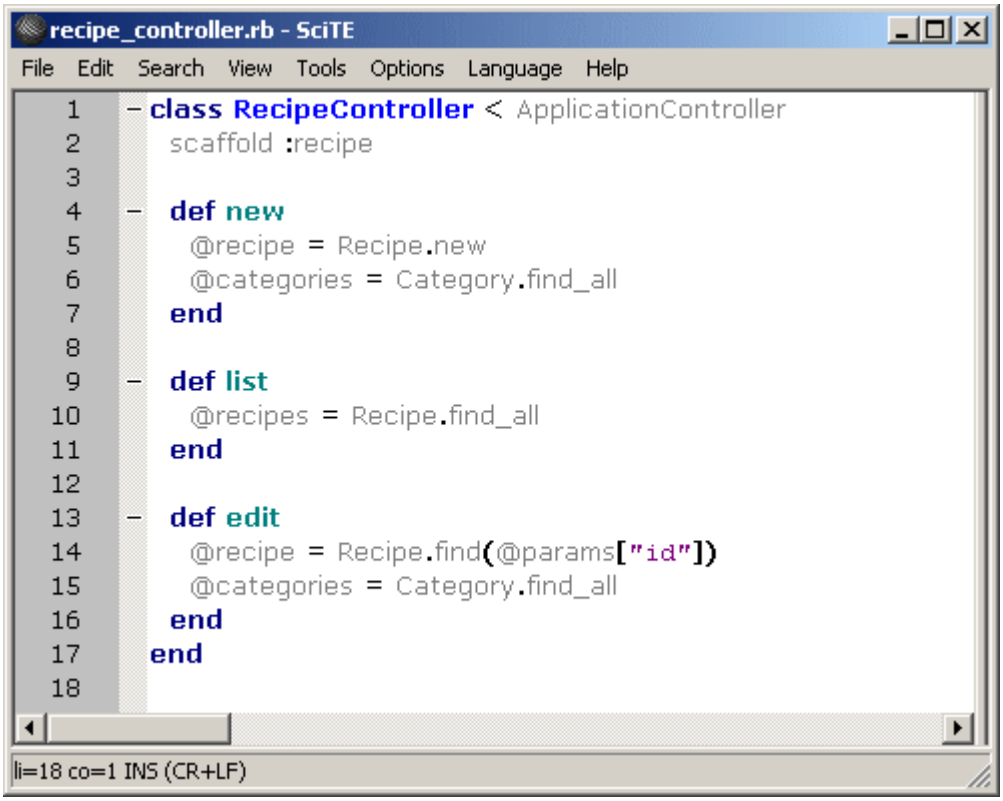


Figure 2. The `Recipe` controller's `new` method

The code `@recipe = Recipe.new` creates a new, empty recipe object and assigns it to the instance variable `@recipe`. Remember, an instance of the Recipe class represents a row in the `recipes` database table. When creating a new recipe object, the Recipe class can assign default values for each field that the view template can use.

The Recipe model class doesn't currently set any such default values, but the view template I'll show off momentarily will use whatever is in the `@recipe` object to initialize the display form. Later, you could add default values in the Recipe class that will show up when you create a new recipe.

As with the edit action, this also retrieves a collection of all categories so that it can display a drop-down list of categories from which the user can choose. The `@categories` instance variable holds this list of categories.

In the directory *c:\rails\cookbook\app\views\recipe*, create a file named *new.rhtml* that contains the HTML template shown below. It's mostly standard HTML, with some extra code to create the `<select>` and `<option>` tags for the drop-down

list of categories:

```
<html>
 <head>
 <title>New Recipe</title>
 </head>
 <body>
 <h1>New Recipe</h1>
 <form action="/recipe/create" method="post">
 <p>
 <b>Title</b><br/>
 <input id="recipe_title" name="recipe[title]" size="30" type="text" value=""/>
 </p>
 <p>
 <b>Description</b><br/>
 <input id="recipe_description" name="recipe[description]"
    size="30" type="text" value=""/>
 </p>
 <p>
 <b>Category:</b><br/>
 <select name="recipe[category_id]">
 <% @categories.each do |category| %>
 <option value="<%= category.id %>">
 <%= category.name %>
 </option>
 <% end %>
 </select>
 </p>
 <p>
 <b>Instructions</b><br/>
 <textarea cols="40" id="recipe_instructions" name="recipe[instructions]"
    rows="20" wrap="virtual">
 </textarea>
 </p>
 <input type="submit" value="Create"/>
 </form>
 <a href="/recipe/list">Back</a>
 </body>
</html>
```

This is not much different from the edit template from Part 1. I left out the recipe's date because I'll set it to the current date when a user posts the form back to the web app. This ensures that the recipe's date will always be its creation date.

If you look at the form tag, you will see that this form will post to a `create` action in the `recipe` controller. Edit *c:\rails\cookbook\app\controllers\recipe_controller.rb* and add this `create` method:

```
def create
    @recipe = Recipe.new(@params['recipe'])
    @recipe.date = Date.today
    if @recipe.save
        redirect_to :action => 'list'
    else
        render_action 'new'
    end
end
```

This method first creates a new recipe object and initializes it from the parameters posted by the form in *new.rhtml*. Then it sets the recipe's date to today's date, and tells the recipe object to save itself to the database. If the save is successful, it redirects to the list action that displays all recipes. If the save fails, it redirects back to the new action so the user can try again.

Give it a try. Start the web server by opening a command window, navigating to *c:\rails\cookbook*, and running the command `ruby script\server`. Then browse to `http://127.0.0.1:3000/recipe/new` and add a new recipe like the one shown in Figure 3.

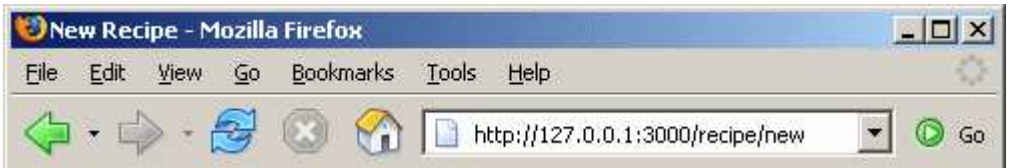


Figure 3. Adding a new recipe with a category

After you create the new recipe, you should see something like Figure 4.

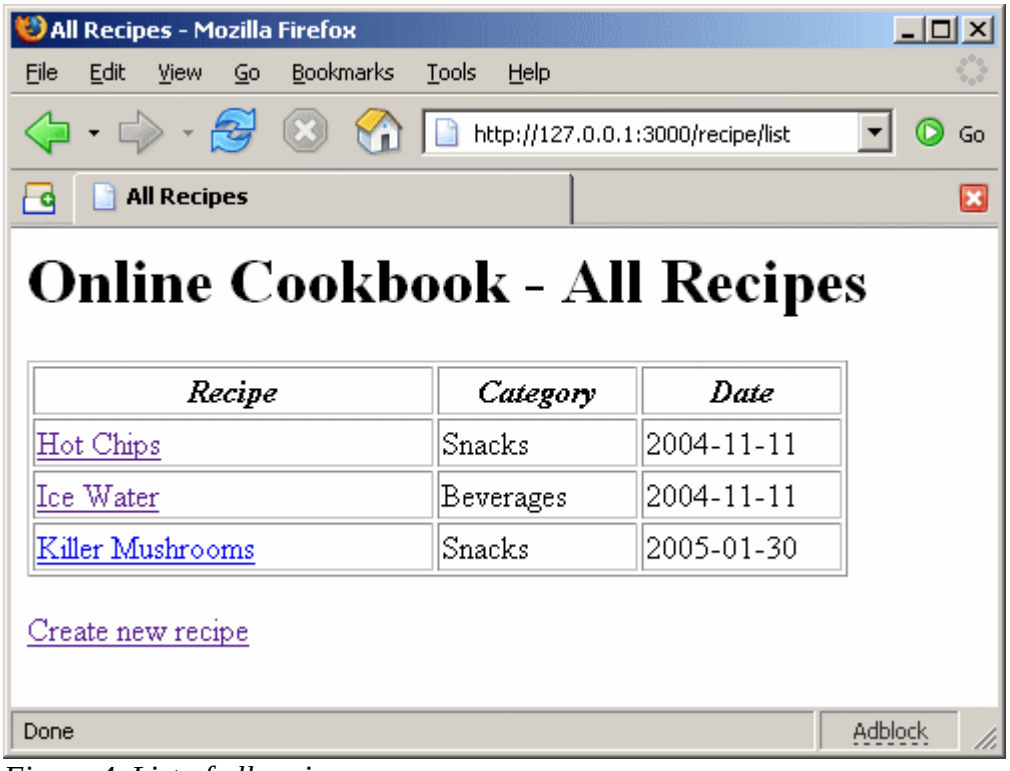


Figure 4. List of all recipes

Deleting a recipe

If you remember from Part 1, once I took over the list action from the scaffolding I no longer had a way to delete a recipe.

The list action must implement this. I'm going to add a small delete link after the name of each recipe on the main list page that will delete its associated recipe when clicked. This is easy.

First, edit *c:\rails\cookbook\app\views\recipe\list.rhtml* and add the delete link by making it look like this:

```
<html>
 <head>
   <title>All Recipes</title>
 </head>
 <body>
   <h1>Online Cookbook - All Recipes</h1>
   <table border="1">
    <tr>
      <td width="40%"><p align="center"><i><b>Recipe</b></i></td>
      <td width="20%"><p align="center"><i><b>Category</b></i></td>
      <td width="20%"><p align="center"><i><b>Date</b></i></td>
    </tr>
    <% @recipes.each do |recipe| %>
      <tr>
        <td>
        <%= link_to recipe.title,
                    :action => "show",
                    :id => recipe.id %>
        <font size=-1>

        <%= link_to "(delete)",
                    {:action => "delete", :id => recipe.id},
                    :confirm => "Really delete #{recipe.title}?" %>
        </font>
        </td>
        <td><%= recipe.category.name %></td>
        <td><%= recipe.date %></td>
      </tr>
    <% end %>
   </table>
 <p><%= link_to "Create new recipe", :action => "new" %></p>
 </body>
</html>
```

The main change here is the addition of this link:

```
<%= link_to "(delete)", {:action => "delete", :id
=> recipe.id},


:confirm => "Really delete #{recipe.title}?" %>
```

This is different from the previous ones. It uses an option that generates a JavaScript confirmation dialog. If the user clicks on OK in this dialog, it follows the link. It takes no action if the user clicks on Cancel.

Try it out by browsing to `http://127.0.0.1:3000/recipe/list`. Try to delete the Ice Water recipe, but click on Cancel when the dialog pops up. You should see something like Figure 5.

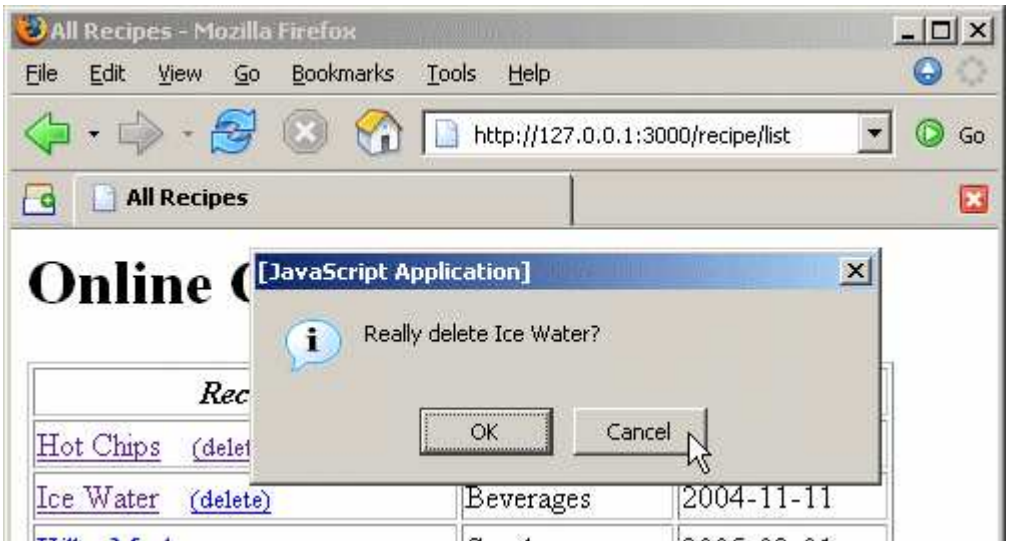


Figure 5. Confirm deleting the Ice Water recipe

Now try it again, but this time click on OK. Did you see the results shown in Figure 6?

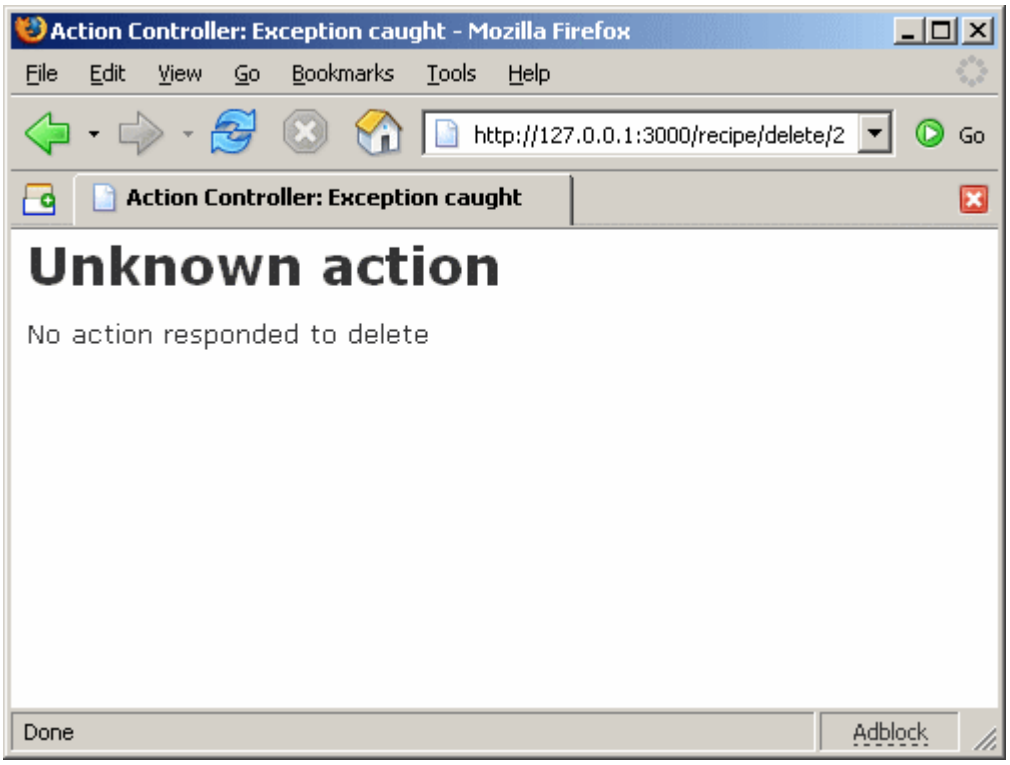


Figure 6. Error deleting the Ice Water recipe

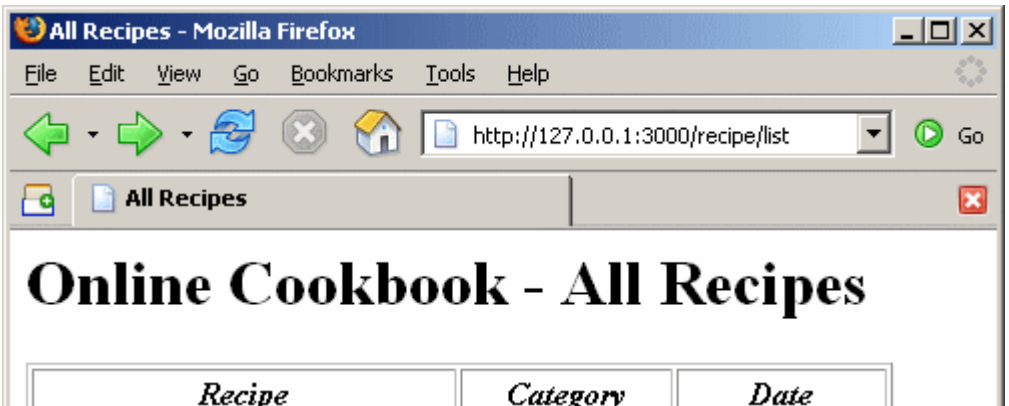
Alright, I admit it; I did this on purpose to remind you that it's OK to make mistakes. I added a link to a delete action in the view template, but never created a delete action in the recipe controller.

Edit *c:\rails\cookbook\app\controllers\recipe_controller.rb* and add this `delete` method:

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

The first line of this method finds the recipe with the ID from the link, then calls the destroy method on that recipe. The second line merely redirects back to the list action.

Try it again. Browse to `http://127.0.0.1:3000/recipe/list` and try to delete the *Ice Water* recipe. Now it should look like Figure 7, and the Ice Water recipe should be gone.



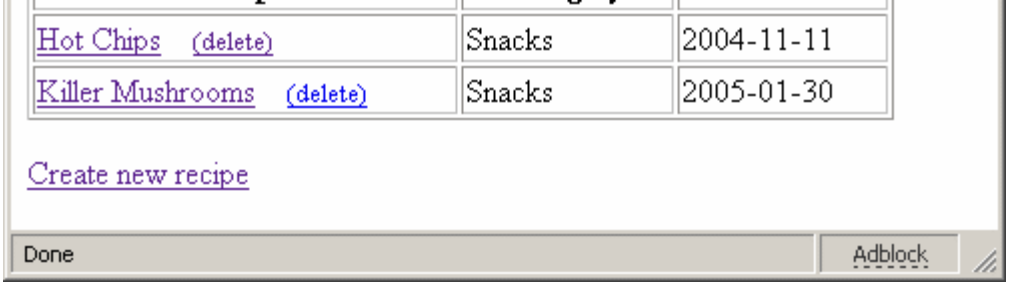


Figure 7. Ice Water recipe is gone

Using layouts

Part 1 used Rails' scaffolding to provide the full range of CRUD operations for categories, but I didn't have to create any links from our main recipe list page. Instead of just throwing in a link on the recipe list page, I want to do something more generally useful: create a set of useful links that will appear at the bottom of every page. Rails has a feature called *layouts*, which is designed just for things like this.

Most web sites that have common headers and footers across all of the pages do so by having each page "include" special header and footer text. Rails layouts reverse this pattern by having the layout file "include" the page content. This is easier to see than to describe.

Edit *c:\rails\cookbook\app\controllers\recipe_controller.rb* and add the `layout` line immediately after the class definition, as shown in Figure 8.

Figure 8. Adding a layout to the recipe controller

This tells the recipe controller to use the file *standard-layout.rhtml* as the layout for all pages rendered by the recipe controller. Rails will look for this file using the path *c:\rails\cookbook\app\views\layouts\standard-layout.rhtml*, but you will have to create the *layouts* directory because it doesn't yet exist. Create this layout file with the following contents:

```
<html>
 <head>
   <title>Online Cookbook</title>
 </head>
 <body>
   <h1>Online Cookbook</h1>
   <%= @content_for_layout %>
   <p>
     <%= link_to "Create new recipe",
                 :controller => "recipe",
                 :action => "new" %>

   <%= link_to "Show all recipes",
               :controller => "recipe",
               :action => "list" %>

   <%= link_to "Show all categories",
               :controller => "category",
               :action => "list" %>
   </p>
 </body>
</html>
```

Only one thing makes this different from any of the other view templates created so far--the line:

```
<%= @content_for_layout %>
```

This is the location at which to insert the content rendered by each recipe action into the layout template. Also, notice that I

have used links that specify both the controller and the action. (Before, the controller defaulted to the currently executing controller.) This was necessary for the link to the category list page, but I could have used the short form on the other two links.

Before you try this out, you must perform one more step. The previous recipe view templates contain some HTML tags that are now in the layout, so edit *c:\rails\cookbook\app\views\recipe\list.rhtml* and delete the extraneous lines at the beginning and end to make it look like this:

```
<table border="1">
 <tr>
   <td width="40%"><p align="center"><i><b>Recipe</b></i></td>
   <td width="20%"><p align="center"><i><b>Category</b></i></td>
   <td width="20%"><p align="center"><i><b>Date</b></i></td>
 </tr>
 <% @recipes.each do |recipe| %>
   <tr>
     <td>
     <%= link_to recipe.title,
                 :action => "show",
                 :id => recipe.id %>
     <font size=-1>

     <%= link_to "(delete)",
                 {:action => "delete", :id => recipe.id},
                 :confirm => "Really delete #{recipe.title}?" %>
     </font>
     </td>
     <td><%= recipe.category.name %></td>
     <td><%= recipe.date %></td>
   </tr>
 <% end %>
</table>
```

Similarly, edit both *c:\rails\cookbook\app\views\recipe\edit.rhtml* and *c:\rails\cookbook\app\views\recipe\new.rhtml* to delete the same extraneous lines. Only the form tags and everything in between should remain.

Browse to `http://127.0.0.1:3000/recipe/list`, and it should look like Figure 9.

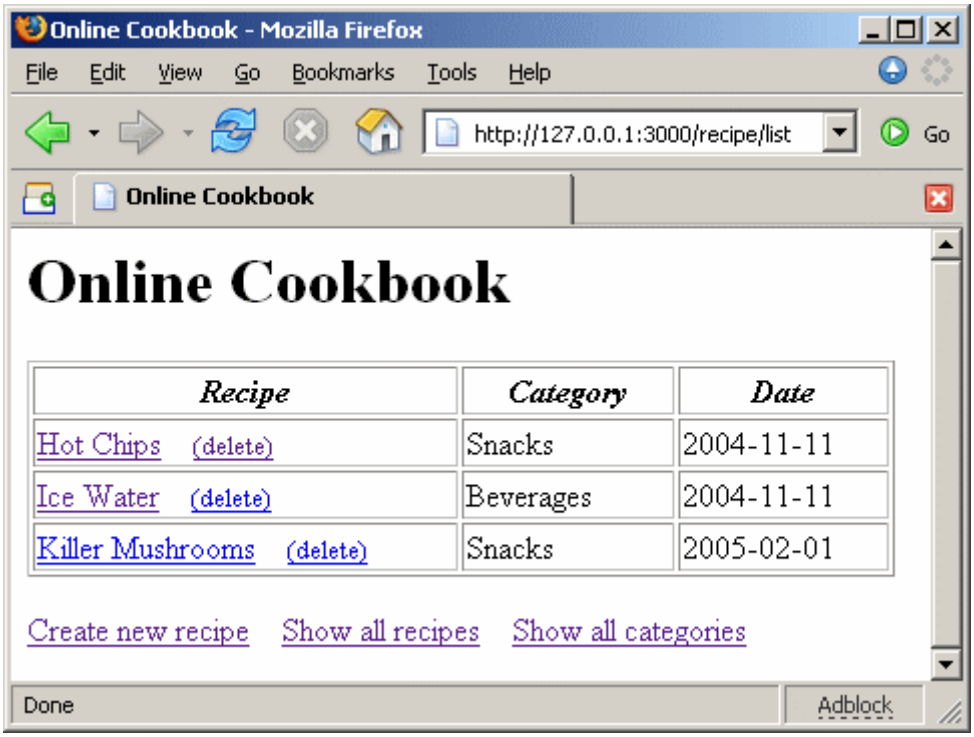


Figure 9. Using a layout with common links

The three links at the bottom of the page should now appear on every page displayed by the recipe controller. Go ahead and try it out!

If you clicked on the "Show all categories" link, you probably noticed that these nice new links did not appear. That is

because the category pages display through the category controller, and only the recipe controller knows to use the new layout.

To fix that, edit *c:\rails\cookbook\app\controllers\category_controller.rb* and add the `layout` line as shown in Figure 10.

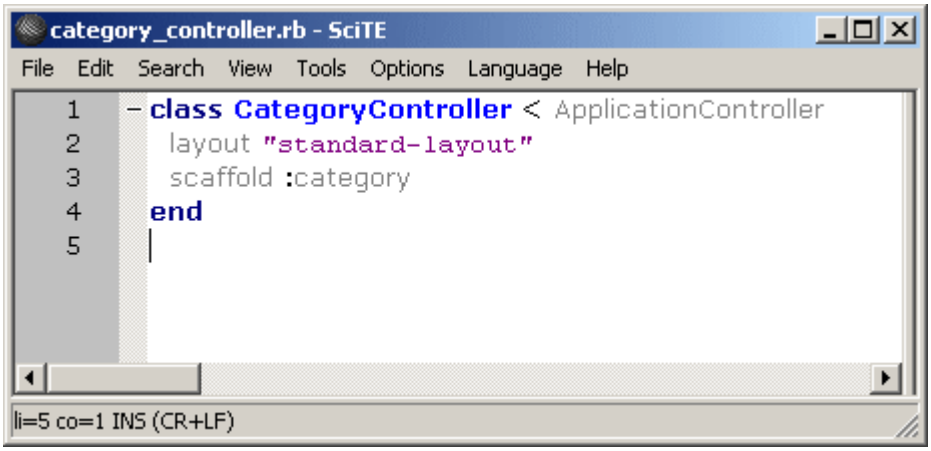


Figure 10. Adding a layout to the category controller

Now you should see the common links at the bottom of all pages of the recipe web application.

Showing recipes in a category

The final task is to add the ability to display only those recipes in a particular category. I'll take the category displayed with each recipe on the main page and turn it into a link that will display only the recipes in that category.

To do this, I'll change the recipe list view template to accept a URL parameter that specifies what category to display, or all categories if the user has omitted the parameter. First, I need to change the `list` action method to retrieve this parameter for use by the view template.

Edit *c:\rails\cookbook\app\controllers\recipe_controller.rb* and modify the `list` method to look like this:

```
def list
    @category = @params['category']
    @recipes = Recipe.find_all
end
```

Then edit *c:\rails\cookbook\app\views\recipe\list.rhtml* to look like this:

```
<table border="1">
 <tr>
   <td width="40%"><p align="center"><i><b>Recipe</b></i></td>
   <td width="20%"><p align="center"><i><b>Category</b></i></td>
   <td width="20%"><p align="center"><i><b>Date</b></i></td>
 </tr>

 <% @recipes.each do |recipe| %>
   <% if (@category == nil) || (@category == recipe.category.name)%>
     <tr>
      <td>
        <%= link_to recipe.title,
                    :action => "show",
                    :id => recipe.id %>
        <font size=-1>

        <%= link_to "(delete)",
                    {:action => "delete", :id => recipe.id},
                    :confirm => "Really delete #{recipe.title}?" %>
        </font>
      </td>
      <td>
        <%= link_to recipe.category.name,
                    :action => "list",
                                    :category => "#{recipe.category.name}" %>
      </td>
```

```
      <td><%= recipe.date %></td>
    </tr>
  <% end %>
<% end %>
</table>
```

There are two changes in here that do all the work. First, this line:

```
<% if (@category == nil) || (@category == recipe.category.name)%>
```

decides whether to display the current recipe in the loop. If the category is `nil` (there was no category parameter on the URL), or if the category from the URL parameter matches the current recipe's category, it displays that recipe.

Second, this line:

```
<%= link_to recipe.category.name,
    :action => "list",
    :category => "#{recipe.category.name}" %>
```

creates a link back to the list action that includes the proper category parameter.

Browse to http://127.0.0.1:3000/recipe/list and click on one of the Snacks links. It should look like Figure 11.

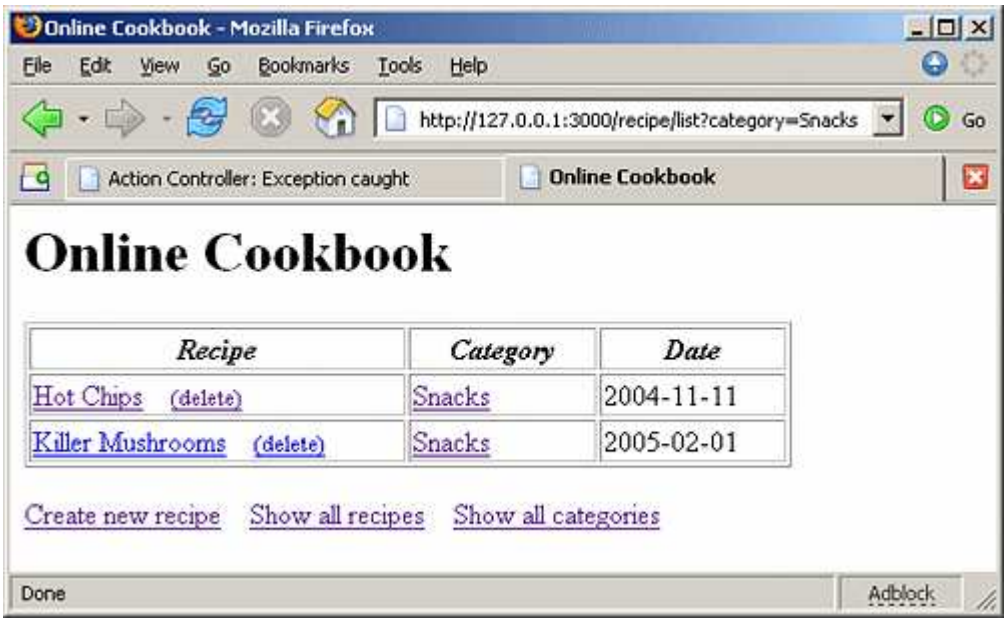


Figure 11. Showing only snacks

What is it? How long did it take?

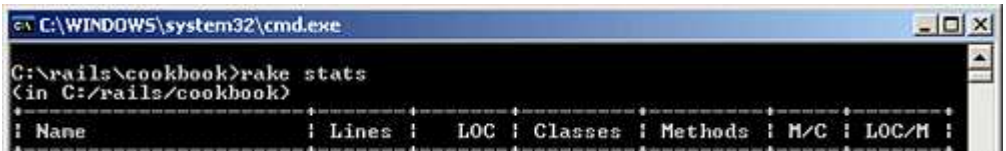
That's it! This is a reasonably functional online cookbook application developed in record time. It's a functional skeleton just begging for polish.

Wading through all of the words and screenshots in this article may have obscured (at least somewhat) exactly what this code can do and in what amount of developer time. Let me present some statistics to try to put it all into perspective.

Fortunately, Rails has some built-in facilities to help answer these questions. Open up a command window in the cookbook directory (*c:\rails\cookbook*) and run the command:

```
rake stats
```

Your results should be similar to Figure 12. Note that LOC means "lines of code."



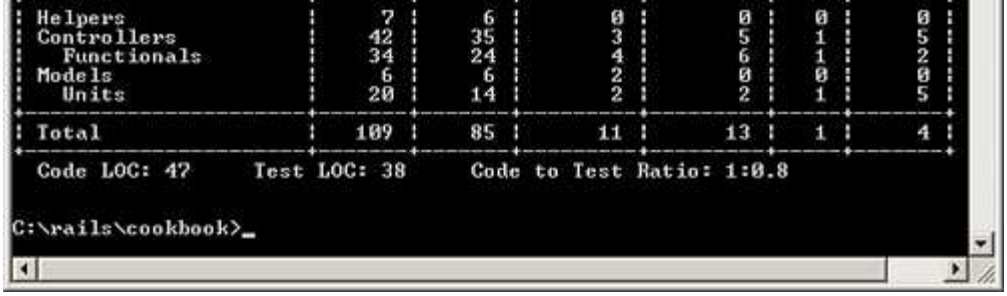


Figure 12. Viewing development statistics

I won't give a detailed description of each number produced, but the last line has the main figure I want to point out:

```
Code LOC: 47
```

This says that the actual number of lines of code in this application (not counting comments or test code) is 47. It took me about 30 minutes to create this application! I could not have come even close to this level of productivity in any other web app development framework that I have used.

Maybe you're thinking that this is an isolated experience using an admittedly trivial example. Maybe you're thinking that this might be OK for small stuff, but it could never scale. If you harbor any such doubts, the next section should lay those to rest.

Ruby on Rails Success Stories

Rails is a relatively young framework. As of this writing, it's been barely six months since the first public release. Yet it debuted with such a stunning feature set and solid stability that a vibrant developer community quickly sprang up around it. Within this time frame, several production web applications have been deployed that were built with Ruby on Rails.

Basecamp

From the site itself:

> Basecamp is a web-based tool that lets you manage projects (or simply ideas) and quickly create client/project extranets. It lets you and your clients (or just your own internal team) keep your conversations, ideas, schedules, to-do lists, and more in a password-protected central location.

Basecamp was the first commercial web site powered by Ruby on Rails. David Heinemeier Hansson, the author of Rails, developed it. At its deployment, it contained 4,000 lines of code with two months of development by a single developer. In fall 2004, Basecamp stated that it had passed the 10,000-user mark. It considers the actual number of registered users to be proprietary information, but the home page currently states that it has "tens of thousands" of users.

43 Things

43 Things is a goal-setting social software web application. It currently has 6,000 registered users and hundreds of thousands of unregistered visitors. 43 Things has 4,500 lines of code that were developed in three months by three full-time developers.

Ta-da Lists

Ta-da Lists is a free online service that implements simple, sharable to-do lists. It features a highly responsive user interface that uses `XMLHttpRequest` to minimize waiting for the server. Ta-da Lists came from one developer using one week of development time producing 579 lines of code.

Snow Devil

Snow Devil is an e-commerce site specializing in snowboards and related equipment. It opened for business only recently, so there is no usage information available at this time. However, it comprises 6,000 lines of code created by two developers in four months.

CD Baby Rewrite

CD Baby is a very successful e-tailer of independent music. In business since 1998, it lists 82,443 artists that together have

sold 1.2 million CDs, paying $12 million back into the artists' pockets.

The CD Baby web site previously involved an increasingly unmanageable 90,000 lines of PHP code. Its authors are in the process of rewriting it in Ruby on Rails. It's too early to find any development information, but the owner of CD Baby is publicly blogging about the process and progress of the conversion.

What does this all mean?

When all is said and done, good design will be more important than the framework in determining how your application performs. Think carefully about your database design and how its tables are indexed. Analyze your data access patterns and consider some strategic denormalization of data. Look for opportunities to cache preprocessed data.

Rails has a lot of powerful features to make it easy to prototype and develop applications quickly, which will leave you with more time to think about your application's features and how to tune it for performance.

A Smattering of Ruby on Rails' Features

Rails has many features that I have not used in this two-part article. I'd like to mention a few of them (with links to more information) to give you a more rounded view of the Rails toolkit.

Caching

Caching is cheap way to speed up your application by saving the results of previous processing (calculations, renderings, database calls, and so on) so as to skip the processing entirely next time. Rails provide three types of caching, in varying levels of granularity:

- page caching
- action caching
- fragment caching

Validation and callbacks

To make sure your data is correct and complete before writing it to the database, you must validate it. Rails has a simple mechanism that allows your web application to validate a data object's data before the object updates or creates the appropriate fields in the database. Read the validation how-to or go straight to the validation API documentation.

ActiveRecord callbacks are hooks into the life cycle of a data object that can trigger logic before or after an operation that alters the state of the data object.

Transactions

ActiveRecord also supports transactions. Quoted straight from the documentation:

> Transactions are protective blocks where SQL statements are only permanent if they can all succeed as one atomic action. The classic example is a transfer between two accounts where you can only have a deposit if the withdrawal succeeded and vice versa. Transaction enforce the integrity of the database and guards the data against program errors or database break-downs. So basically you should use transaction blocks whenever you have a number of statements that must be executed together or not at all.

For example, consider the code:

```
transaction do
    david.withdrawal(100)
    mary.deposit(100)
end
```

Testing

Rails was built with testing in mind and provides support for testing your web application. An extensive online tutorial shows how to test a Rails web application.

Generators

Generators are the helper scripts that you can use to generate code for your application. You have already used generators to create new controllers and models, and at the beginning of this article I showed you how to use a new generator to create scaffolding.

Rails also supports user-created add-on generators. For example, Tobias Luetke has written a Login Generator that creates all the code for easily adding authentication, users, and logins to your Rails app.

Security

By now, everyone should know the importance of good security in web applications. The Ruby on Rails web site has an online security manual that describes common security problems in web applications and how to avoid them with Rails.

Parting Thoughts

Rails is not your run-of-the-mill, proof-of-concept web framework. It is the next level in web programming, and the developers who use it will make web applications faster than those who don't; single developers can be as productive as whole teams. Best of all, it's available right now, under an MIT license.

I believe that there hasn't been an improvement in productivity like this in recent programming history.

Editor's note: Want more Ruby on Rails? See Ajax on Rails.

Resources

Web sites

- Official Ruby home page
- Official Ruby on Rails home page
- The best way to learn Ruby: read the free book *Programming Ruby*
- Primary home for open source Ruby projects: RubyForge
- Ruby on Rails open source projects
- Ruby on Rails how-tos
- Four Days on Rails, an excellent next-steps guide to use after "Rolling with Ruby on Rails."

Mailing lists

- Rails mailing list
- Ruby-talk mailing list

Curt Hibbs is a senior software developer in Saint Louis, Missouri, with more than 30 years' experience in platforms, languages, and technologies too numerous to list.

Return to ONLamp.com.