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/2005/01/20/rails.html
See this if you're having trouble printing code examples

Rolling with Ruby on Rails

by Curt Hibbs
01/20/2005

Maybe you've heard about Ruby on Rails, the *super productive* new way to develop web applications, and you'd like to give it a try, but you don't know anything about Ruby or Rails. This article steps through the development of a web application using Rails. It won't teach you how to program in Ruby, but if you already know another object-oriented programming language, you should have no problem following along (and at the end you can find links on learning Ruby).

Let's answer a couple of burning questions before rolling up our sleeves to build a web application!

What is Ruby?

Ruby is a pure object-oriented programming language with a super clean syntax that makes programming elegant and fun. Ruby successfully combines Smalltalk's conceptual elegance, Python's ease of use and learning, and Perl's pragmatism. Ruby originated in Japan in the early 1990s, and has started to become popular worldwide in the past few years as more English language books and documentation have become available.

What is Rails?

Rails is an open source Ruby framework for developing database-backed web applications. What's special about that? There are dozens of frameworks out there and most of them have been around much longer than Rails. Why should you care about yet another framework?

What would you think if I told you that you could develop a web application *at least* ten times faster with Rails than you could with a typical Java framework? You can--*without* making any sacrifices in the quality of your application! How is this possible?

Part of the answer is in the Ruby programming language. Many things that are very simple to do in Ruby are not even possible in most other languages. Rails takes full advantage of this. The rest of the answer is in two of Rail's guiding principles: *less software* and *convention over configuration*.

Less software means you write fewer lines of code to implement your application. Keeping your code small means faster development and fewer bugs, which makes your code easier to understand, maintain, and enhance. Very shortly, you will see how Rails cuts your code burden.

Convention over configuration means an end to verbose XML configuration files--there aren't any in Rails! Instead of configuration files, a Rails application uses a few simple programming conventions that allow it to figure out everything through reflection and discovery. Your application code and your running database already contain everything that Rails needs to know!

Seeing is Believing

We developers often hear the excessive hype that always seems to accompany something new. I can just imagine that skeptical look on your face as you hear my dubious claims. *Ten times faster development, indeed!*

I'm not asking you to accept this on blind faith. I'll show you how to prove it to yourself. First, I'll install the needed

software. Then I will lead you through the development of a web application.

Installing the Software

We'll develop this web application on Windows. You can still follow along if you use a Linux or Macintosh system, but your screen will look different from the screen shots shown below and you will have to install software packages built specifically for your system. See the Resources section at end of this article for additional software links.

To develop this web application, install the following software:

- Ruby
- The Rails framework
- The MySQL database (and the MySQL-Front GUI client)

Step 1: Install Ruby

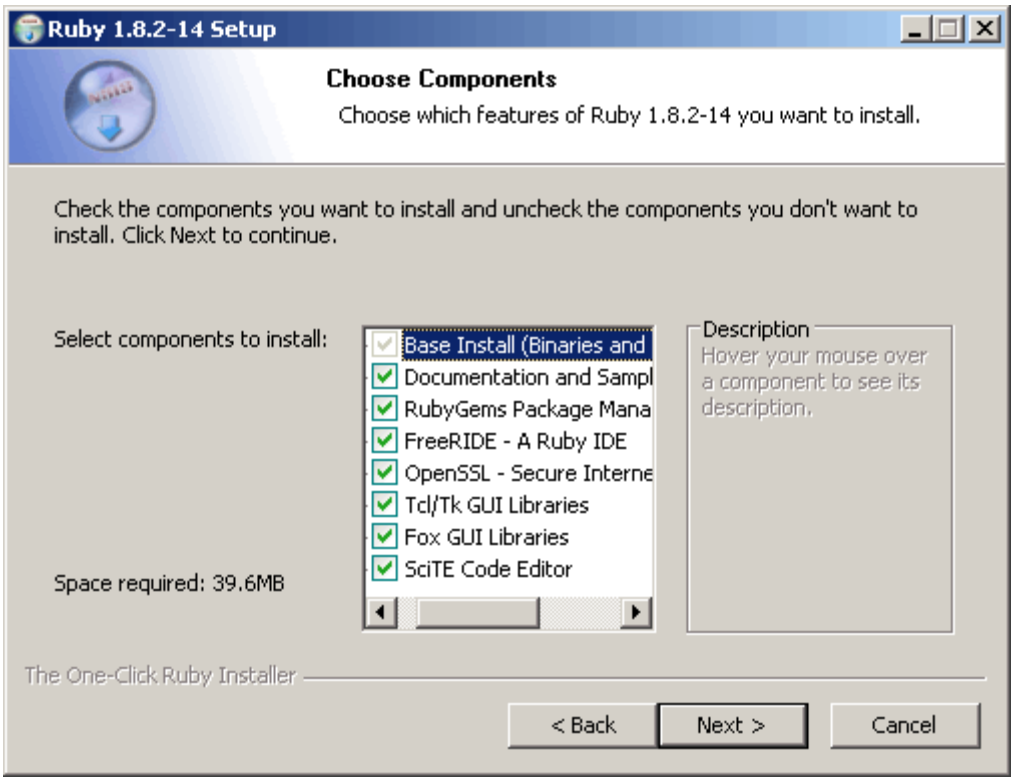


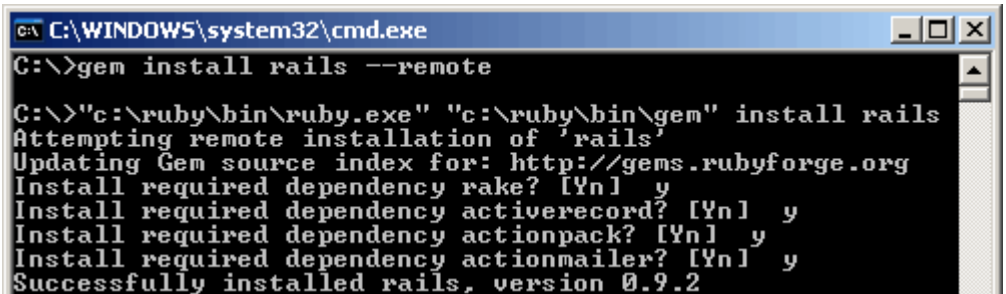
Figure 1. Ruby Windows installer

Installing Ruby couldn't be any simpler:

1. Download the latest One-Click Ruby Installer for Windows (shown in Figure 1). As of this writing, the latest is `ruby182-14.exe`).
2. Double-click on the downloaded executable and follow the installation instructions. Unless you have some special needs, just press `Enter` to accept all of the defaults.

Note to Linux and OS X users: The Windows installer come with the RubyGems package manager already installed. Whatever means you use to install Ruby, you will probably have to install RubyGems, also.

Step 2: Install Rails



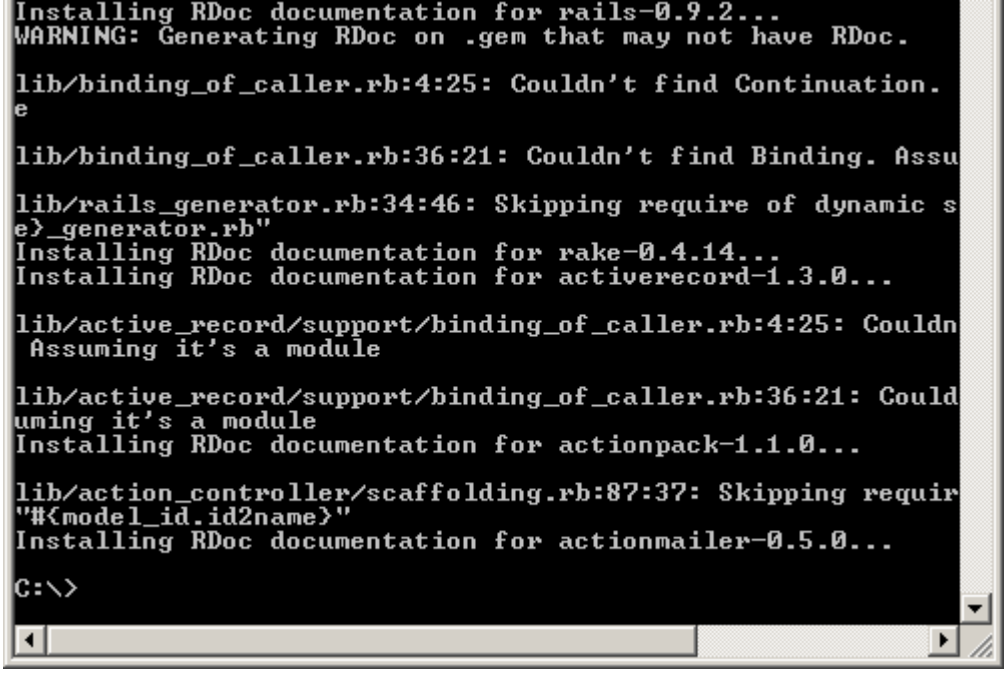


Figure 2. Installing Rails through RubyGems

Now we can use the RubyGems package manager to download and install Rails 0.9.4 (the version covered by this tutorial), as Figure 2 shows:

1. Open a command window and run the command `gem install rails --remote`.
2. RubyGems will also install all of the other libraries that Rails depends on. For each of these dependencies, RubyGems will ask you if you want to install it. Answer "y" (yes) to each one.

Step 3: Install MySQL

Figure 3. MySQL Server setup wizard

We still need to install our database server. Rails supports many different databases. We'll use MySQL.

1. Download the latest "essential" version of the MySQL Windows installer. (Currently, that is `Windows Essentials (x86) 4.1.7`.)
2. Double-click on the installer (Figure 3) and accept all of the defaults, though skip signing up for a mysql.com account.
3. When you reach the final panel of the installer, clicking the Finish button will bring up the configuration wizard.
4. In the configuration wizard, you can also just accept all of the defaults, except that in the security panel you *must* uncheck the "Modify Security Settings" checkbox (Figure 4). This is because starting with version 4.1.7, MySQL uses a new authentication algorithm that is not compatible with older client software, including the current version of Rails. By unchecking this box, you can access MySQL without a password.

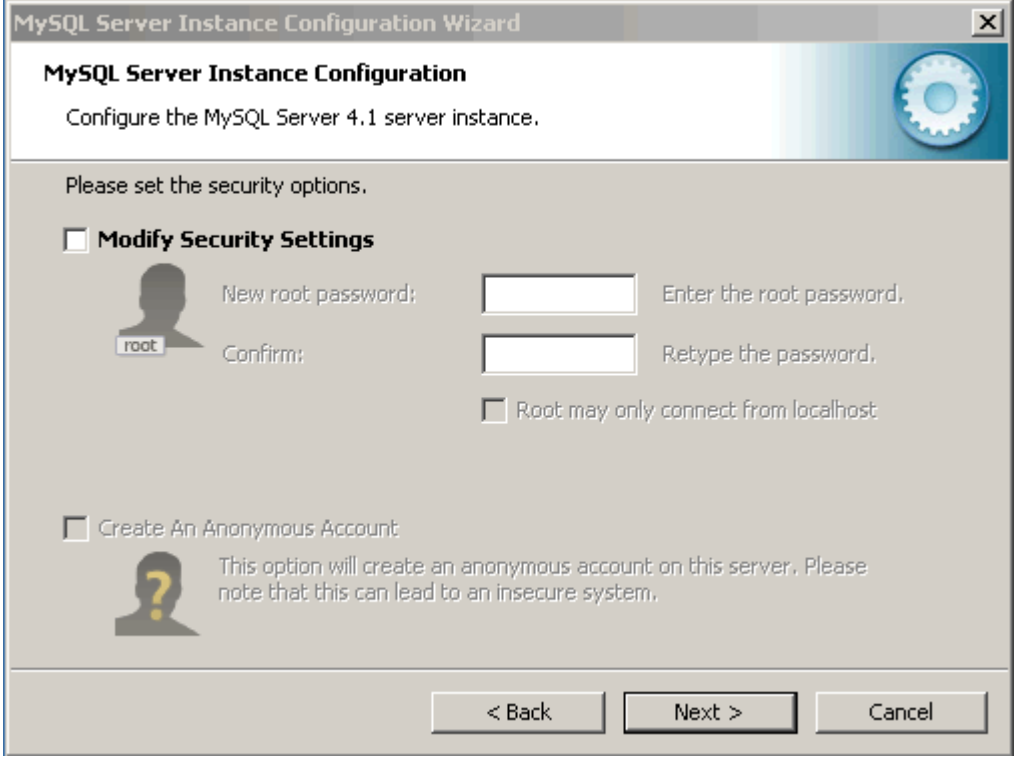


Figure 4. MySQL configuration wizard

Step 4: Install MySQL-Front

MySQL-Front is a graphical interface for the MySQL database. It is an inexpensive commercial application, but you can try it for free for 30 days. In this article, we will use MySQL-Front to develop our database. If you prefer, you can also just send SQL commands to the database from the command line.

1. Download the latest version of MySQL-Front (currently, version 3.1).
2. Double-click on the downloaded executable and follow the installation instructions (Figure 5). Unless you have some special needs, you can accept all of the defaults.

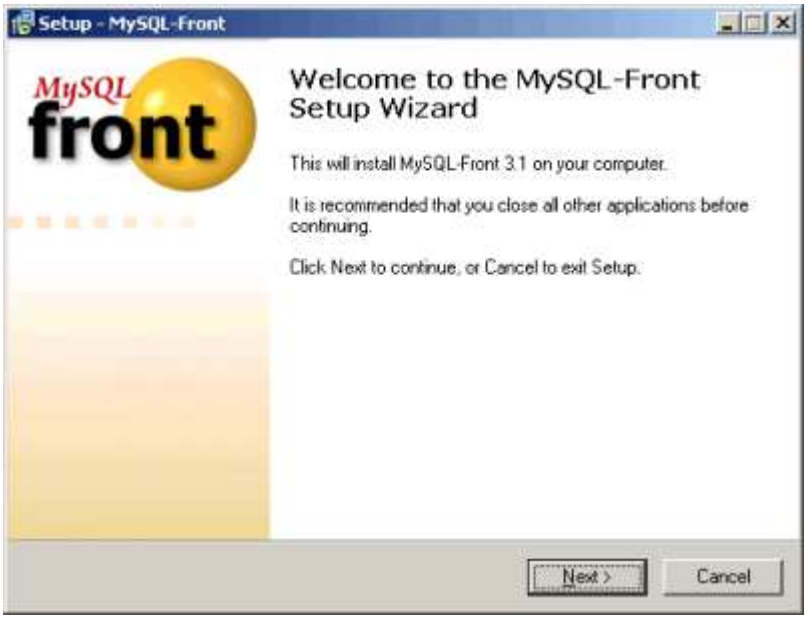


Figure 5. MySQL-Front installer

Let's Write Code

We'll create an online collaborative cookbook for holding and sharing everyone's favorite recipes. We want our cookbook to:

- Display a list of all recipes.
- Create new recipes and edit existing recipes.

- Assign a recipe to a category (like "dessert" or "soup").

You can create your cookbook application in any directory you like, but I used *c:\rails\cookbook*. All paths used in this article assume this base directory path. If you choose a different location, be sure to make the necessary adjustments as you see application paths in this article.

Creating an Empty Rails Web Application

Rails is both a runtime web app framework *and* a set of helper scripts that automate many of the things you do when developing a web application. In this step, we will use one such helper script to create the entire directory structure and the initial set of files to start our cookbook application.

1. Open a command window and navigate to where you want to create this cookbook web application. I used *c:\rails*.
2. Run the command:

   ```
   rails cookbook
   ```

   This will create a *cookbook* subdirectory containing a complete directory tree of folders and files for an empty Rails application.

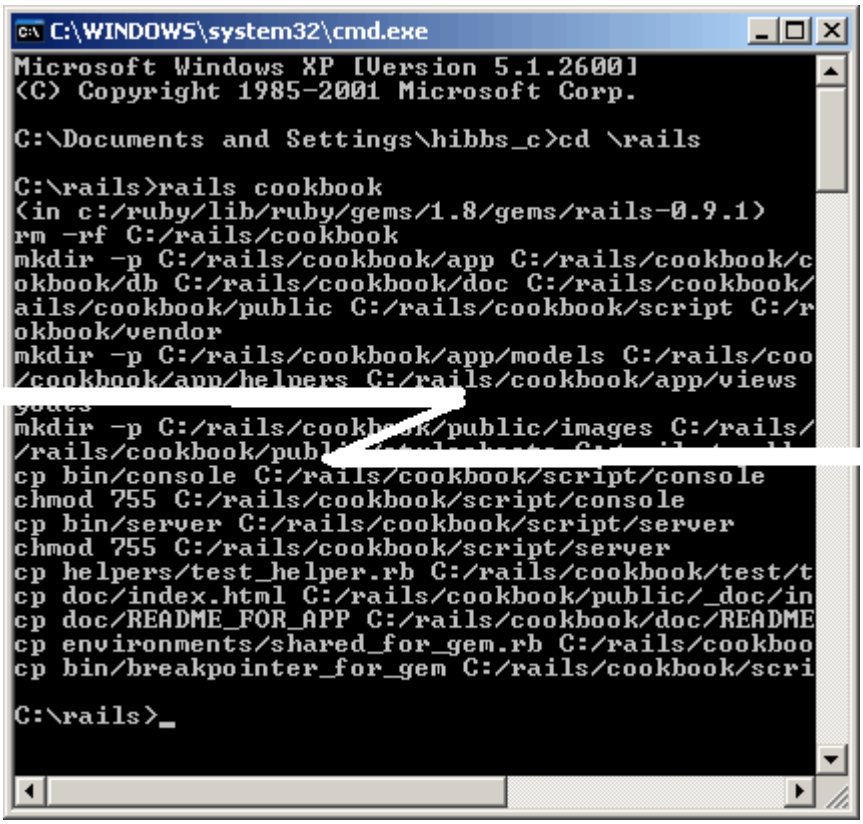


Figure 6. A newly created Rails application directory

Testing the Empty Web Application

A Rails web application can run under virtually any web server, but the most convenient way to develop a Rails web application is to use the built-in WEBrick web server. Let's start this web server and then browse to our cookbook application.

- In your open command window, move into the cookbook directory.
- Run the command:

  ```
  ruby script\server
  ```

  to start the server (Figure 7).

- Now open your browser and browse to `http://127.0.0.1:3000/`. You should see something like Figure 8. *[Editor's note: Unless you're following along with the article, these links probably won't work for you. Don't panic--127.0.0.1 is a special address reserved for the local machine.]*

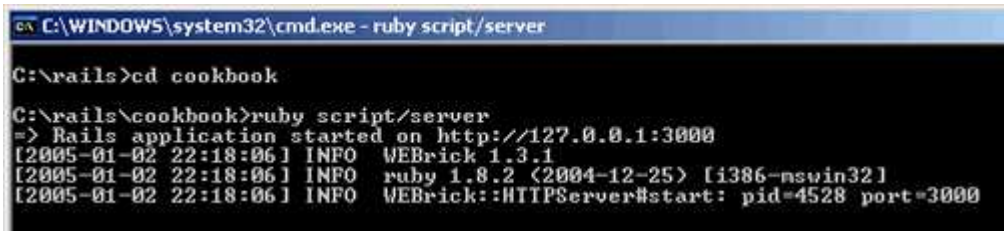


Figure 7. Starting the WEBrick server

Leave the command window open and the web server running, as we will be using it as we proceed.

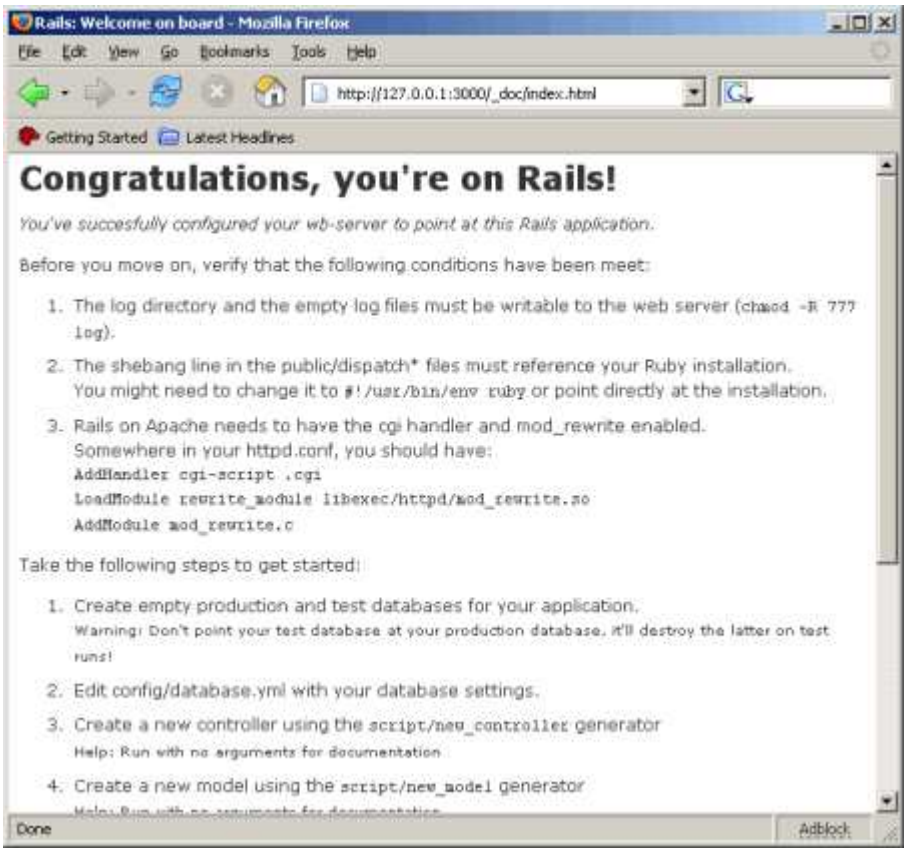
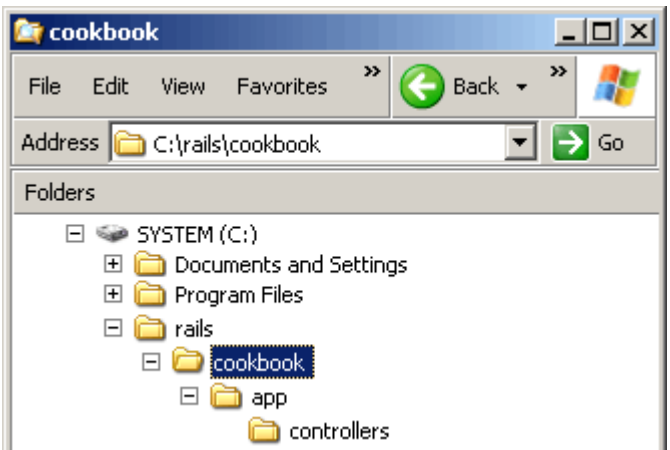


Figure 8. The Rails default page

A Rails Application's Directory Structure

Rails tries very hard to minimize the number of decisions you have to make and to eliminate unnecessary work. When you used the `rails` helper script to create your empty application, it created the entire directory structure for the application (Figure 9). Rails knows where to find things it needs within this structure, so you don't have to tell it. *Remember, no configuration files*!



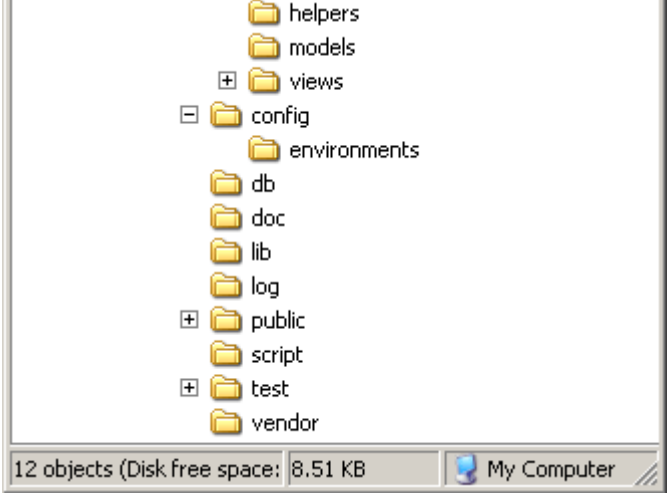


Figure 9. A Rails application directory structure

Most of our development work will be creating and editing files in the *c:\rails\cookbook\app* subdirectories. Here's a quick rundown of how to use them.

- The *controllers* subdirectory is where Rails looks to find controller classes. A controller handles a web request from the user.
- The *views* subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user's browser.
- The *models* subdirectory holds the classes that model and wrap the data stored in our application's database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it *dead simple*!
- The *helpers* subdirectory holds any helper classes used to assist the model, view, and controller classes. This helps to keep the the model, view, and controller code small, focused, and uncluttered.

Controllers and URLs

In a moment, we will create our cookbook database and begin developing our application. First, it's important to understand how controllers work in Rails and how URLs map into (and execute) controller methods.

Controller classes handle web requests from the user. The URL of the request maps to a controller class and a method within the class. How does this work?

Leave your existing command window open with the web server running, open a second command window, and navigate to the application's base directory, *c:\rails\cookbook*. It will look like Figure 10, at least in a moment.

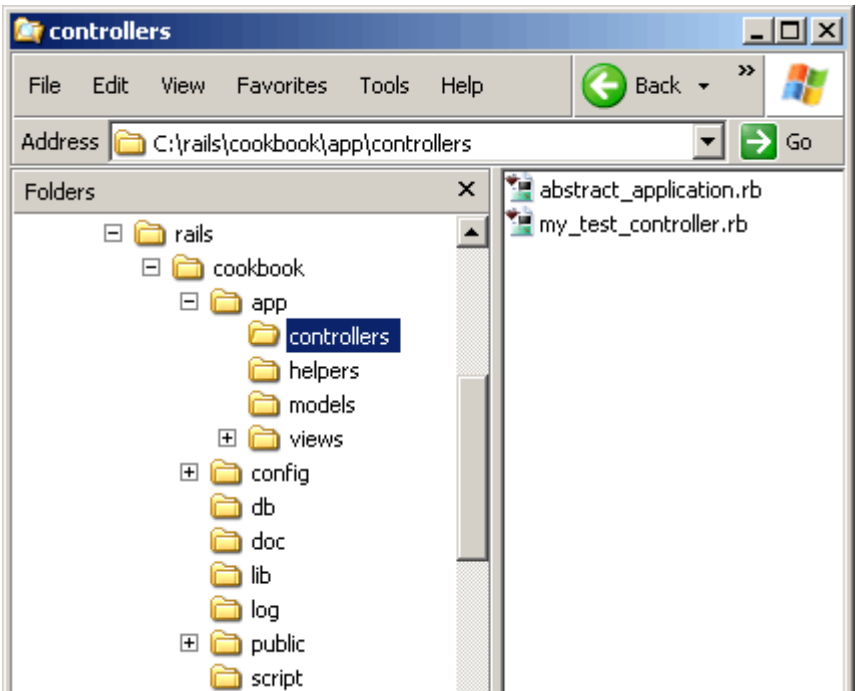


Figure 10. The cookbook controller directory

We will use another Rails helper script to create a new controller class for us. In the command window, run the command:

```
ruby script\generate controller MyTest
```

This will create a file named *my_test_controller.rb* containing a skeleton definition for the class `MyTestController`.

In the *c:\rails\cookbook\controllers* directory, right-click on this file and choose Edit. The file should resemble Figure 11.

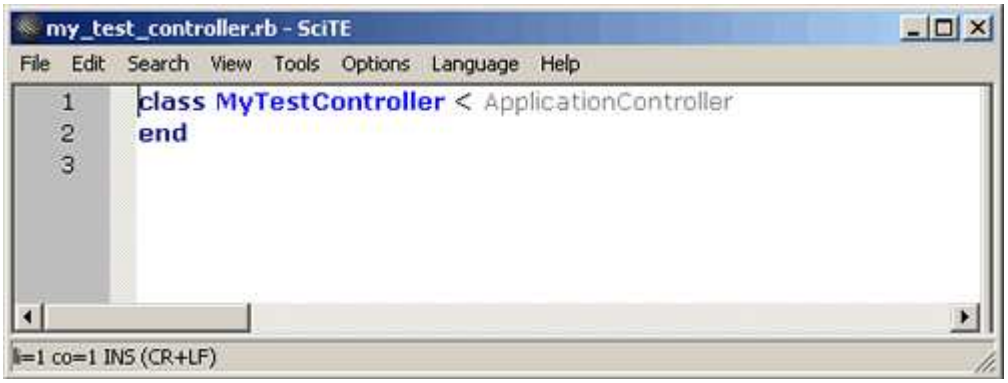


Figure 11. Editing `MyTestController`

What happens if you browse to something that you *know* does not exist? Try `http://127.0.0.1:3000/garbage/`. Figure 12 shows the results.

Figure 12. Browsing to an unknown controller

That's not too surprising. Now try `http://127.0.0.1:3000/My_Test/`, shown in Figure 13.

Figure 13. Browsing to the new controller

Hmmm. Now that's different. The `MyTest` part of the URL maps to the newly created controller. Now it seems that Rails tried to find an action named `index` in this controller but couldn't.

Let's fix that. Add an `index` method to your controller class as in Figure 14.

Figure 14. The `index` method of `MyTestController`

Refresh your browser, and you should now see something more like Figure 15.

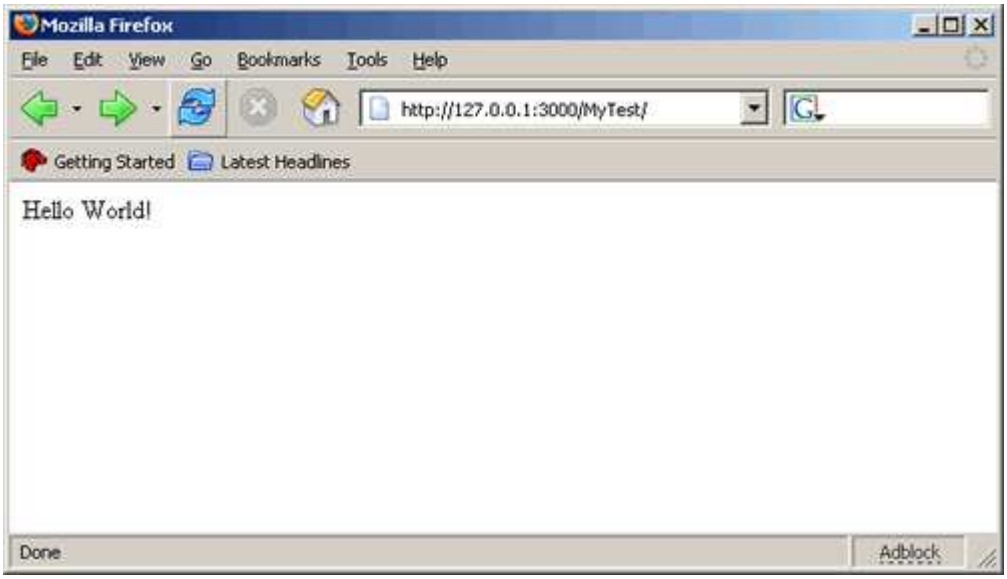


Figure 15. The result of the `index` method

You will have the same results with `http://127.0.0.1:3000/My_Test/index`, too.

Let's add another action to the controller just to make sure you have the idea. Add the `dilbert` method from Figure 16.

Figure 16. The `dilbert` method

Now browse to `http://127.0.0.1:3000/My_Test/dilbert` and you'll see something like Figure 17.

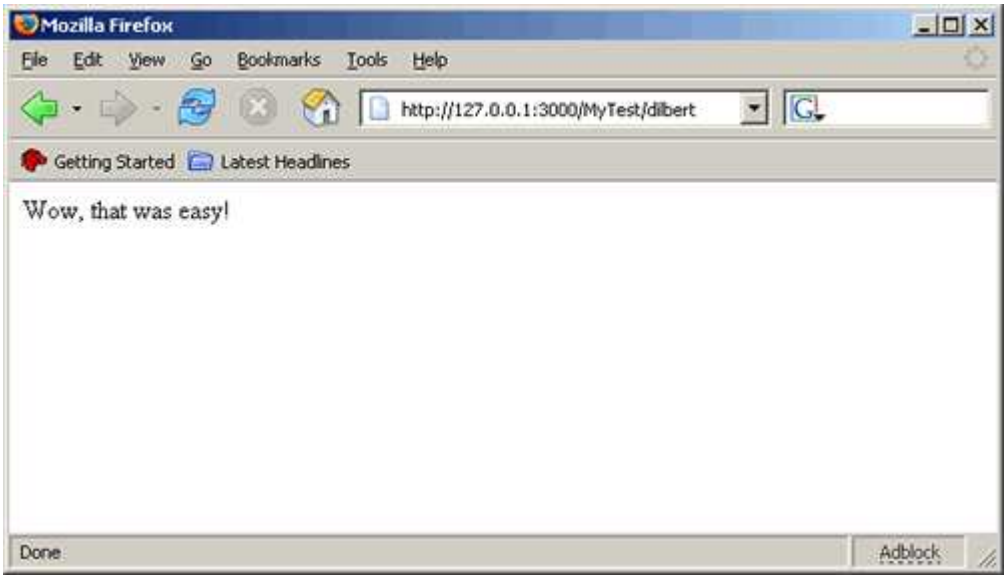


Figure 17. The output of the `dilbert` method

I think you have the idea.

Let's create our database now and work on some real pieces of our cookbook application.

Creating the Cookbook Database

It's time to create the cookbook database and tell Rails how to find it. (This is the *only* configuration that you will find in Rails.)

1. Start MySQL-Front and log in to your locally running MySQL instance (`localhost`) as root, using an empty password. You should see something like Figure 18.

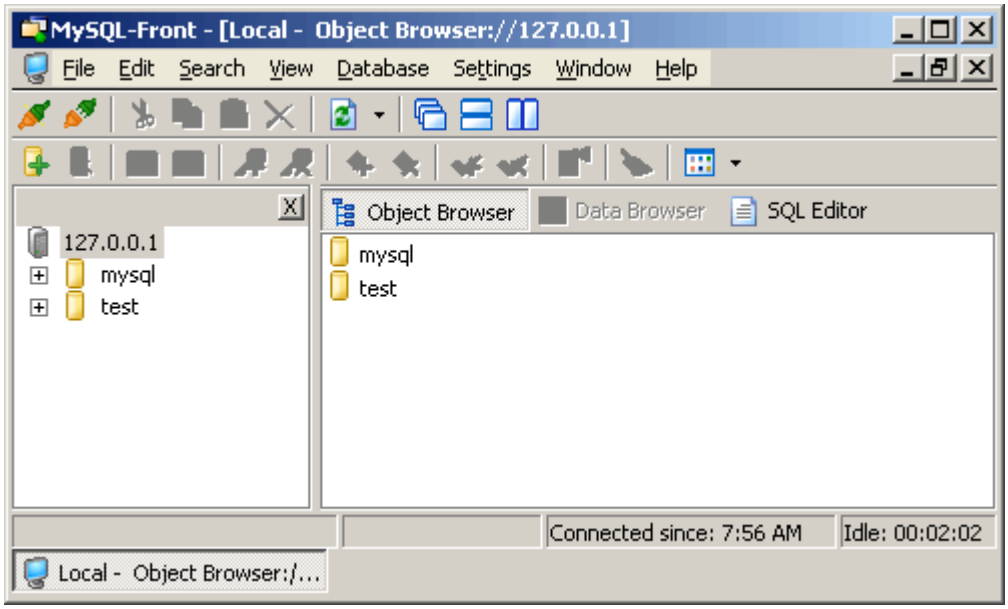


Figure 18. MySQL-Front

2. There are two existing databases, `mysql` and `test`. Create a new database named `cookbook`. Execute the menu command `Database>New>Database...` and enter the database name `cookbook`, as Figure 19 illustrates.

Figure 19. Creating a new database

Click Ok to create the database.

3. To tell Rails how to find the database, edit the file *c:\rails\cookbook\config\database.yml* and change the database name to `cookbook`. Leave the username as root and the password empty. When you finish, it should look something like Figure 20.

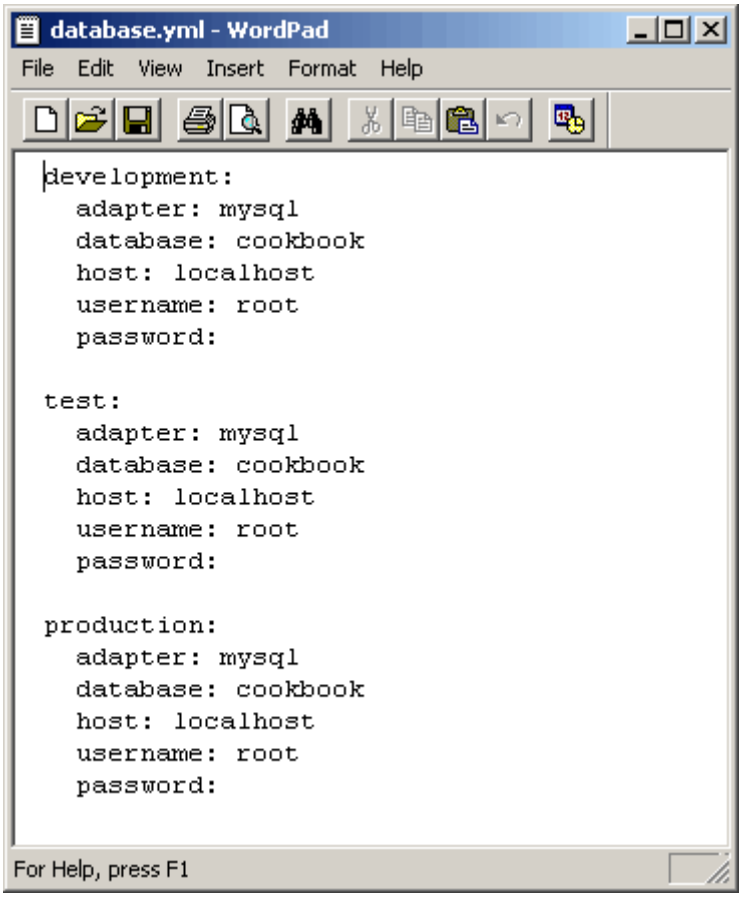


Figure 20. The database.yml configuration file

Rails lets you run in development mode, test mode, or production mode, using different databases. This application uses the same database for each.

Editor's note: A recent change in Rails requires that you restart the webserver, or else Rails will never see the new database and the subsequent steps will fail. Hit Ctrl-C or close the window as appropriate and relaunch the web server at this point.

Creating the `recipes` Table

Our cookbook will contain recipes, so let's create a table in our database to hold them.

In the left-hand pane of MySQL-Front, right click on the cookbook database you just created and select `New>Table...` from the pop-up menu (Figure 21).

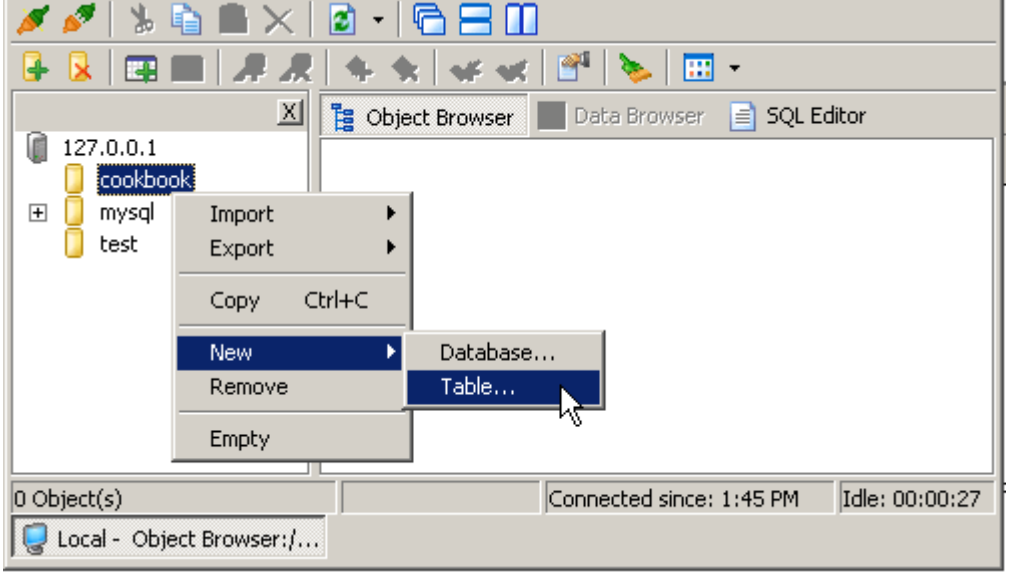


Figure 21. Creating a new table

Name the table `recipes` (Figure 22).

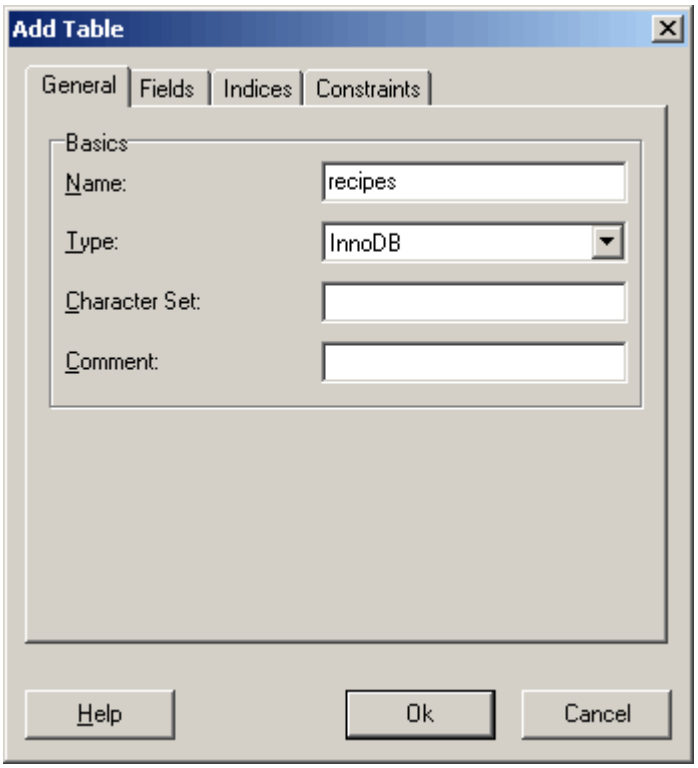
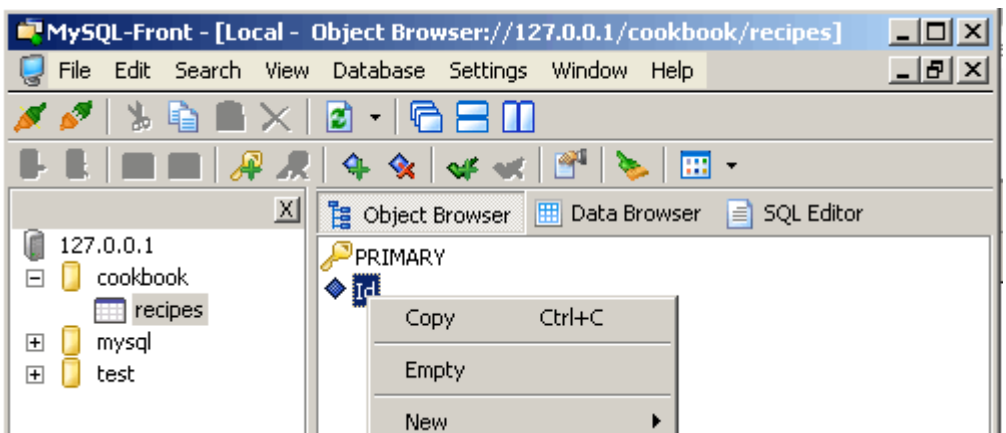


Figure 22. The Add Table dialog box

Important note: MySQL-Front will automatically create a primary key named `Id`, but Rails prefers to call it `id` (all lowercase). I'll explain more later, but for now just change it. In the left pane, select the `recipes` table you just created. In the right pane, right-click on the `Id` field, select Properties (Figure 23), and change the name to `id`.



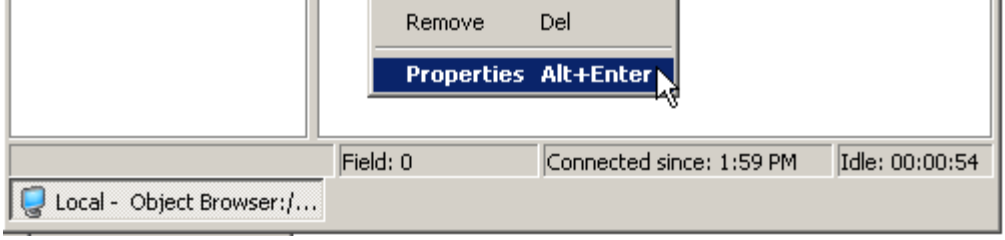


Figure 23. Renaming the primary key

Adding Recipe Fields

Now that we have a recipes table, we can start adding fields (columns) to hold recipe data. Let's start by creating `title` and `instructions` fields. Eventually, we will need more fields, but this is a good place to start.

With the recipe table selected, right click in a blank area of the right pane and select `New>Field...` (Figure 24).

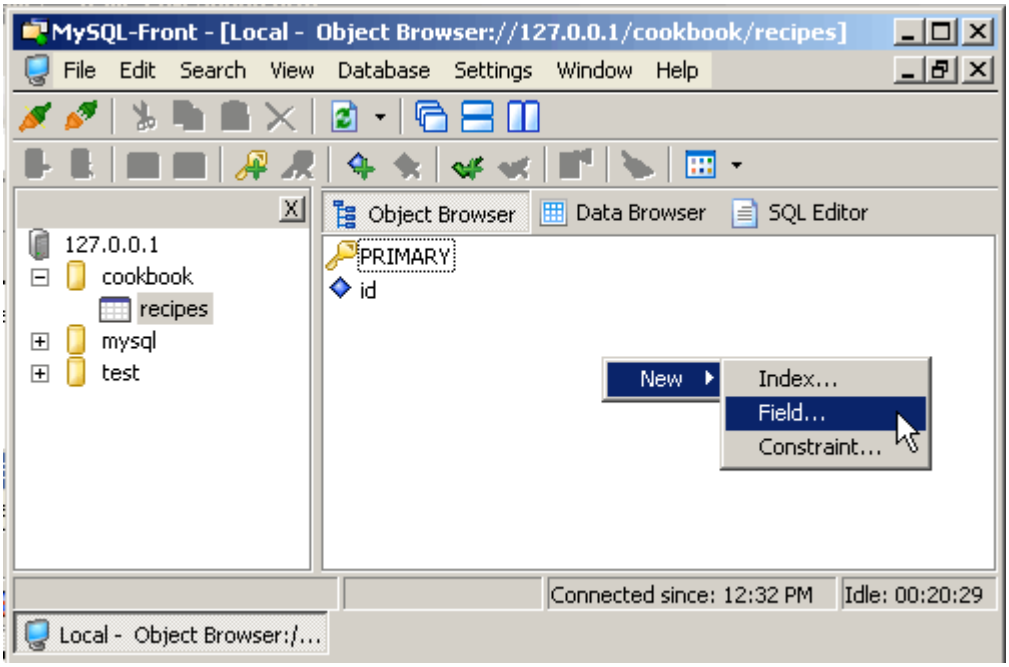
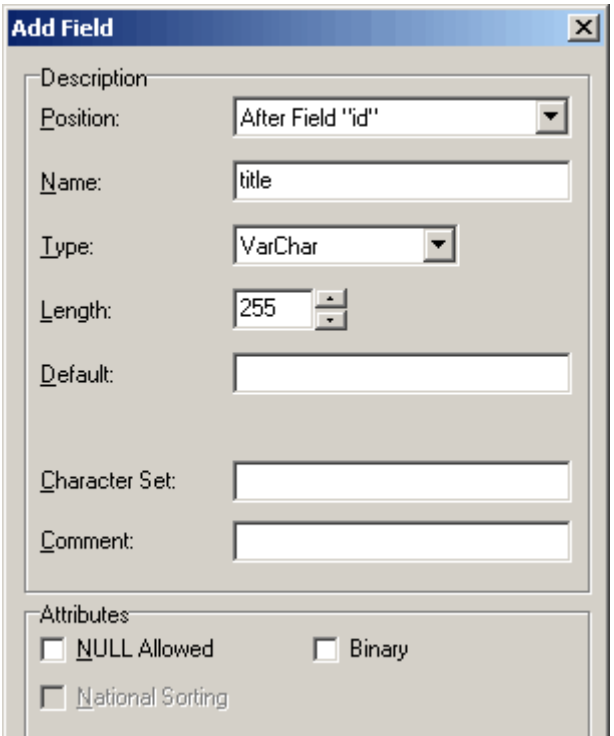


Figure 24. Adding a new field

Create the recipe title field as a `varchar(255)` with nulls *not* allowed, so that every recipe *must* have title text. Figure 25 shows the options in the pop-up window.



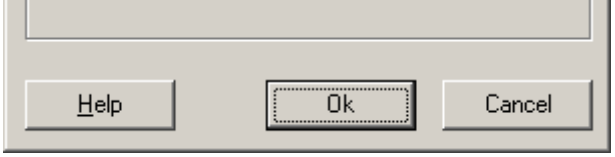


Figure 25. Adding the `title` field

Repeat the above procedure to create an `instructions` field as `text`, as shown in Figure 26.

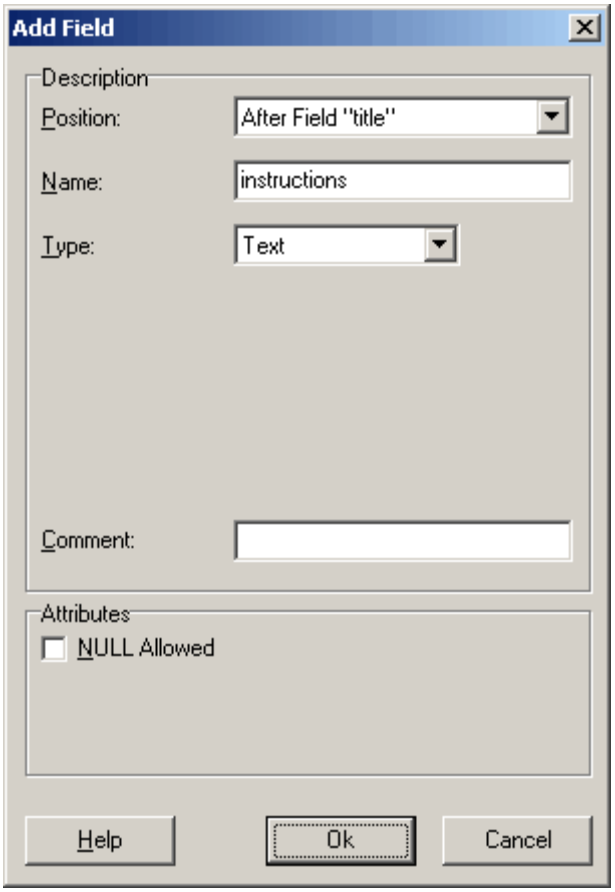


Figure 26. Adding the `instructions` field

The recipe table should now resemble Figure 27.

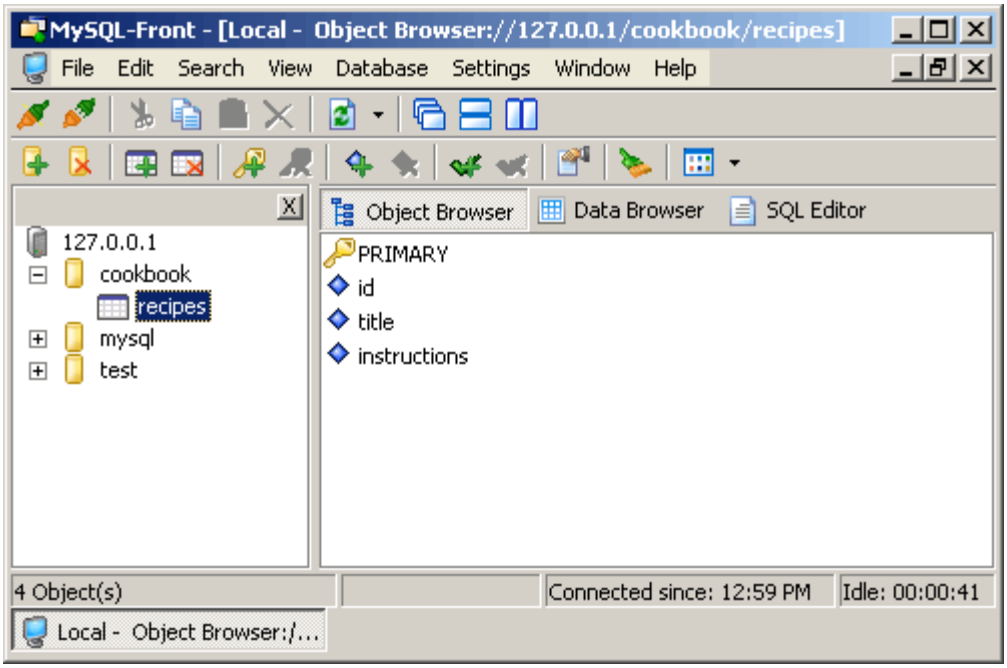


Figure 27. The modified `recipe` table

The Excitement Begins

Everything we have done up to this point has been pretty short and painless, but not particularly exciting. This is where that changes. We can now have the very beginnings of our cookbook application up and running in *record time*!

Create the Model

First, create a `Recipe` model class that will hold data from the `recipes` table in the database. Figure 28 shows where it should live.

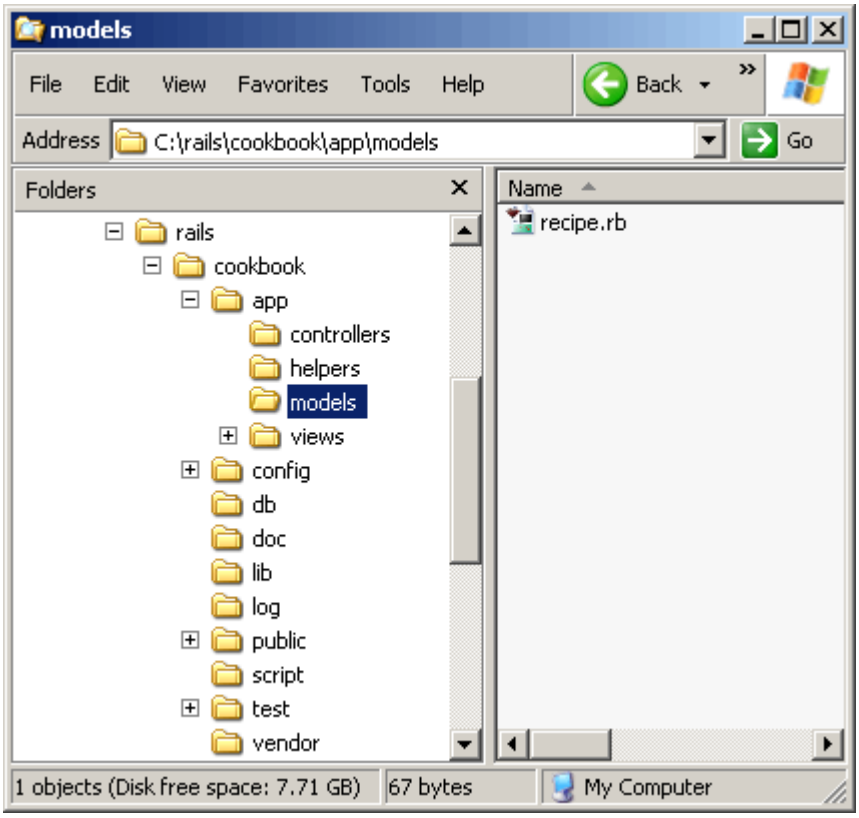


Figure 28. The `Recipe` model class

Open a command window to the cookbook directory (*c:\rails\cookbook*) and run the command:

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

This will create a file named *recipe.rb* containing a skeleton definition for the `Recipe` class. Right-click on this file and choose Edit to look inside (Figure 29).

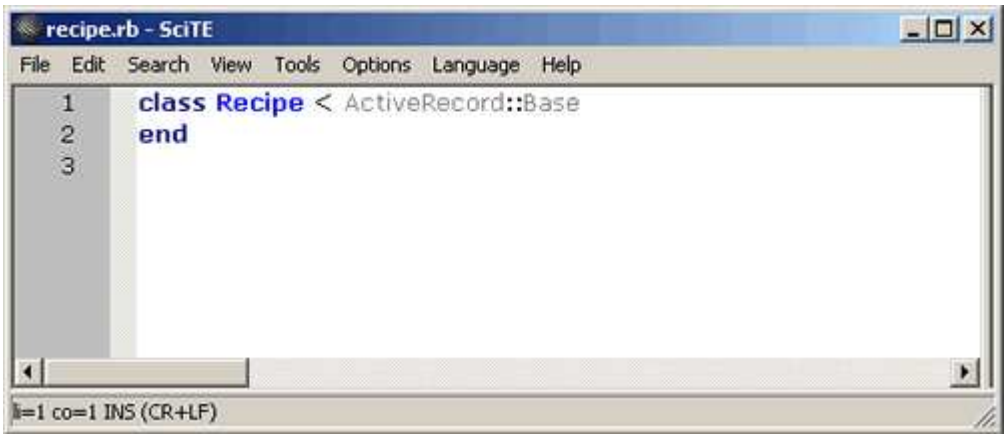


Figure 29. The contents of recipe.rb

This seemingly empty class definition is the recipe business object that Rails maps to the `recipes` table in the database. You will see more concretely what I mean by this in a moment. Right now, I want to point out that this little bit of programming magic happened because we used a Rails naming convention: a singular model class name (`Recipe`) maps to a plural database table (`recipes`). Rails is smart about English pluralization rules, so `Company` maps to `companies`, `Person` maps to `people`, and so forth.

Further, Rails dynamically populates the `Recipe` class with methods for accessing the rows in the `recipes` table and an attribute for each column in the table.

Very shortly, you will see a dramatic demonstration of this dynamic connection between the `Recipe` class and the `recipes` table.

We are now very close to seeing something work. We need to create a recipe controller (Figure 30) with actions to manipulate the recipes in the database via the standard CRUD operations: create, read, update, and delete. Rails makes this easier than you might think.

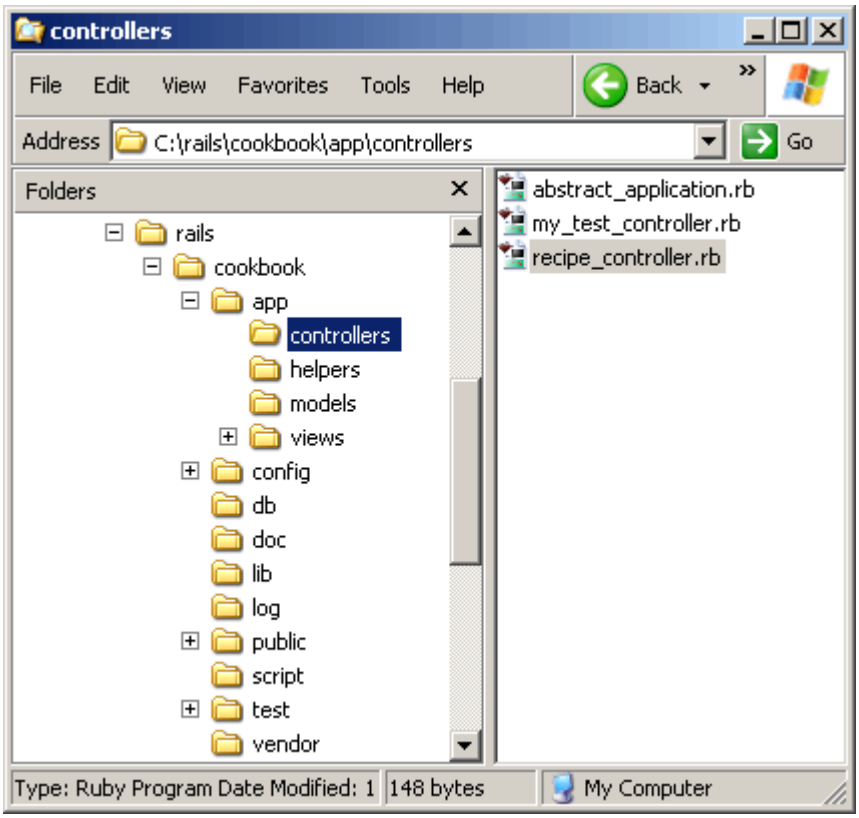


Figure 30. The Recipe controller in its native environment

Open a command window in the cookbook directory (*c:\rails\cookbook*) and run the command:

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

This will create a file named *recipe_controller.rb* containing a skeleton definition for the `RecipeController` class. Right-click on this file, choose Edit, and add the line `scaffold :recipe` as shown in Figure 31.

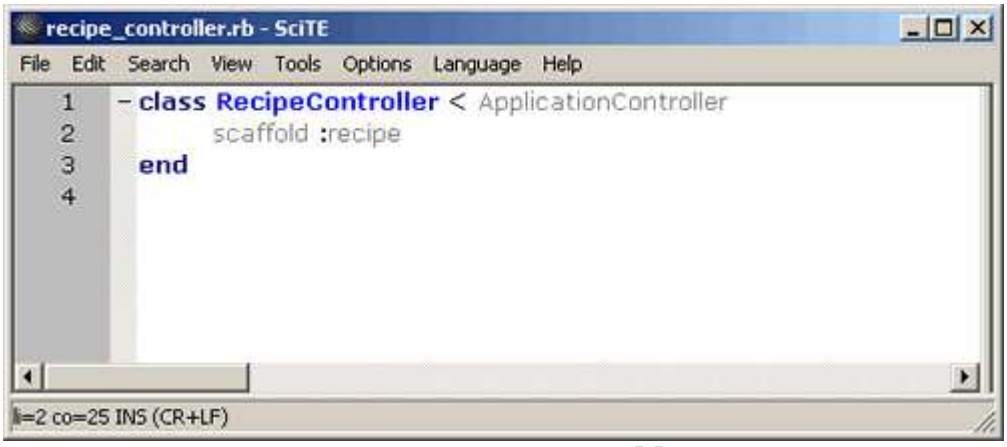


Figure 31. One line of code in `RecipeController`

This single line of code will bring the database table to life. It defines actions for all CRUD operations, immediately allowing us to create, read, update, and delete recipes in our database!

Open a browser and navigate to `http://127.0.0.1:3000/recipe/new`. You should see something like Figure 32.

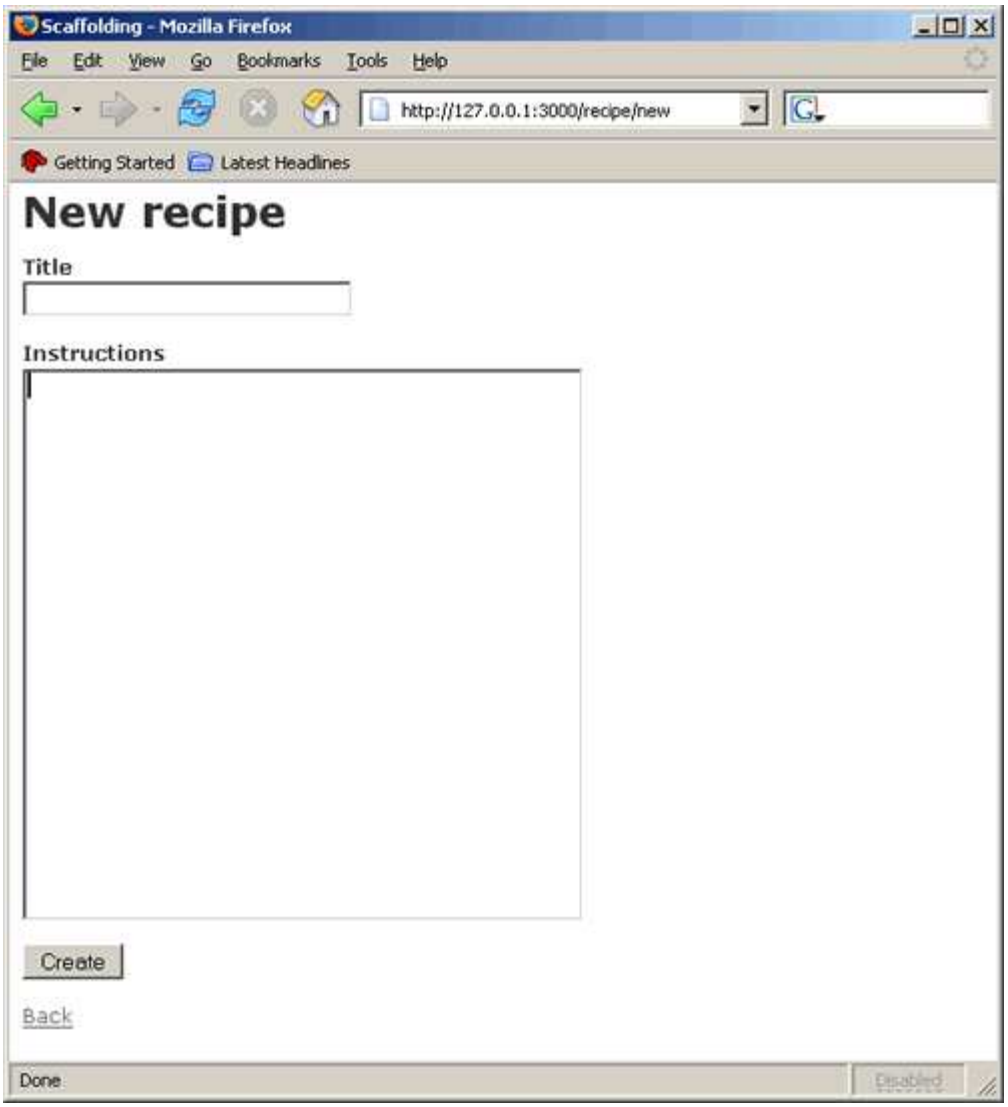
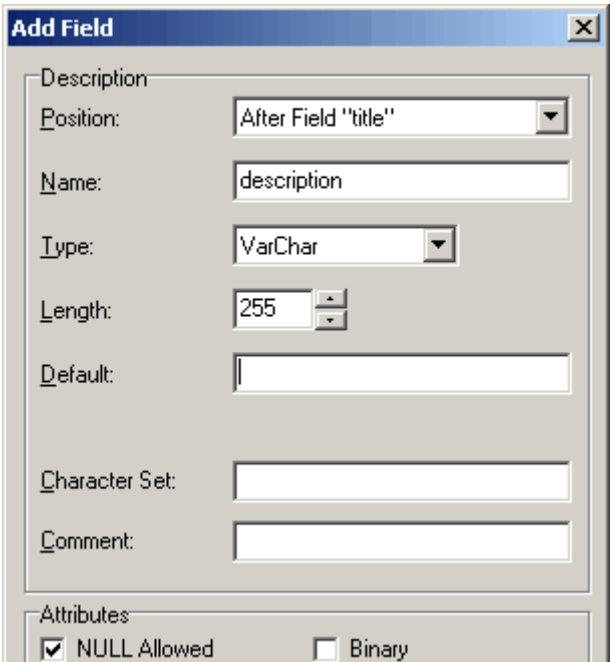


Figure 30. Creating a new recipe page

Now this is pretty cool! We haven't done much of anything and we can already start to populate our database. Don't do that just yet, though. Let's add a few more fields to the `recipe` table first.

Use MySQL-Front to add `description` and `date` fields between the `title` and `instructions` fields (Figures 33 and 34).



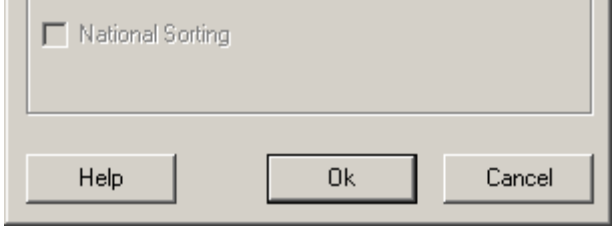


Figure 33. Adding the `description` field

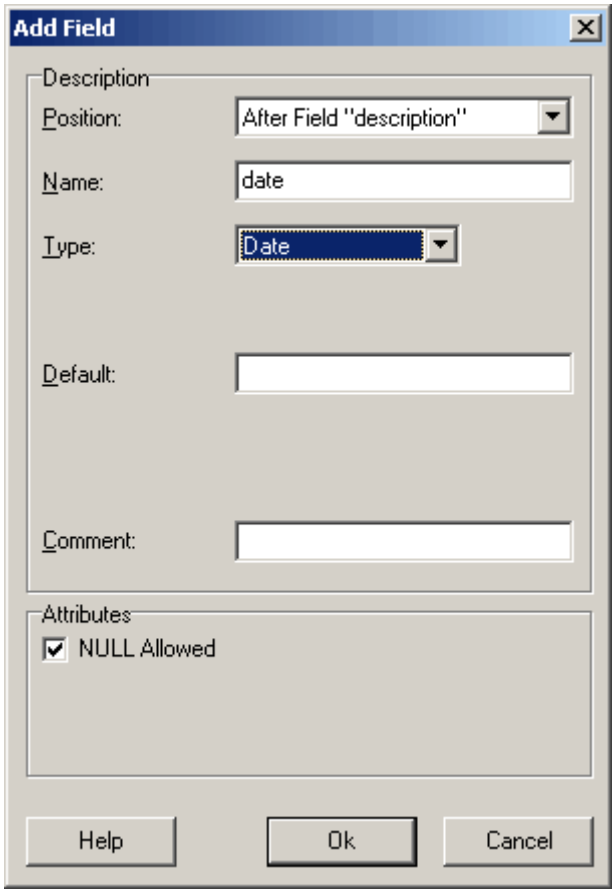
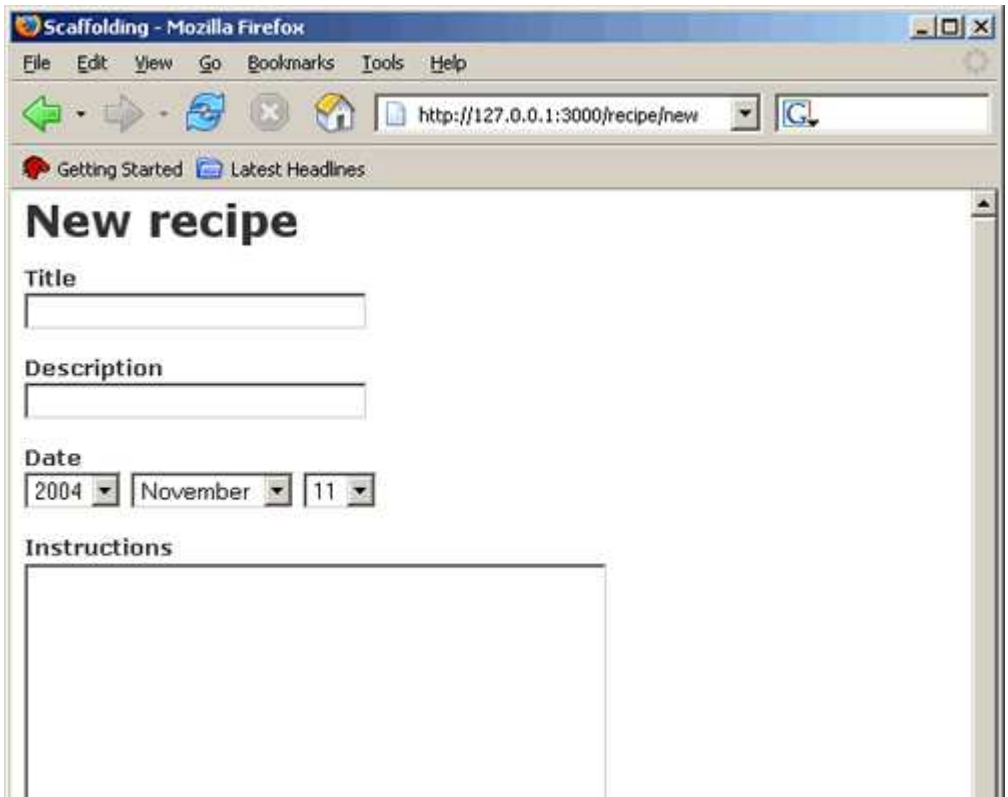


Figure 34. Adding the `date` field

Refresh your browser to see a page similar to Figure 35.



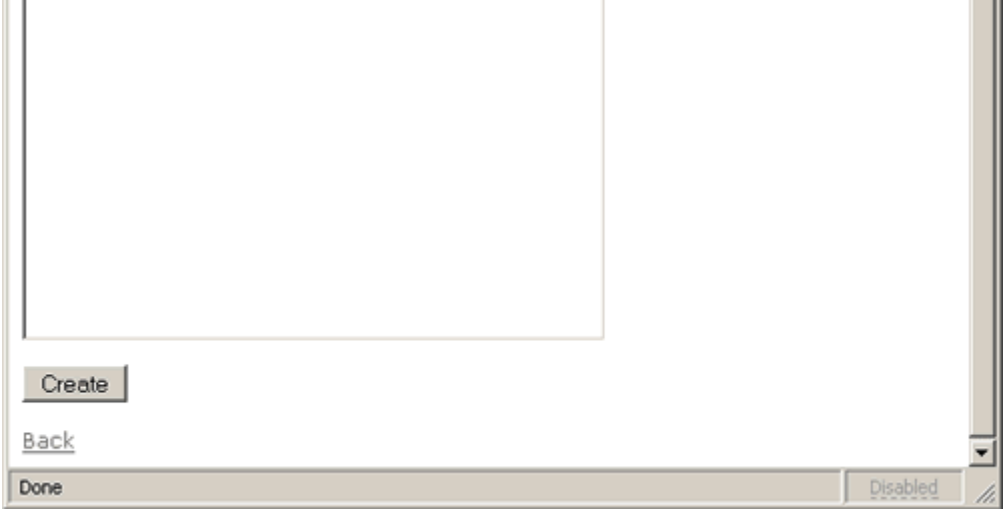


Figure 35. A new recipe page with the new fields

Now, that is way beyond cool--it's awesome!

OK, calm down and enter a test recipe. Fill in the fields as shown in Figure 36 and click on the Create button.

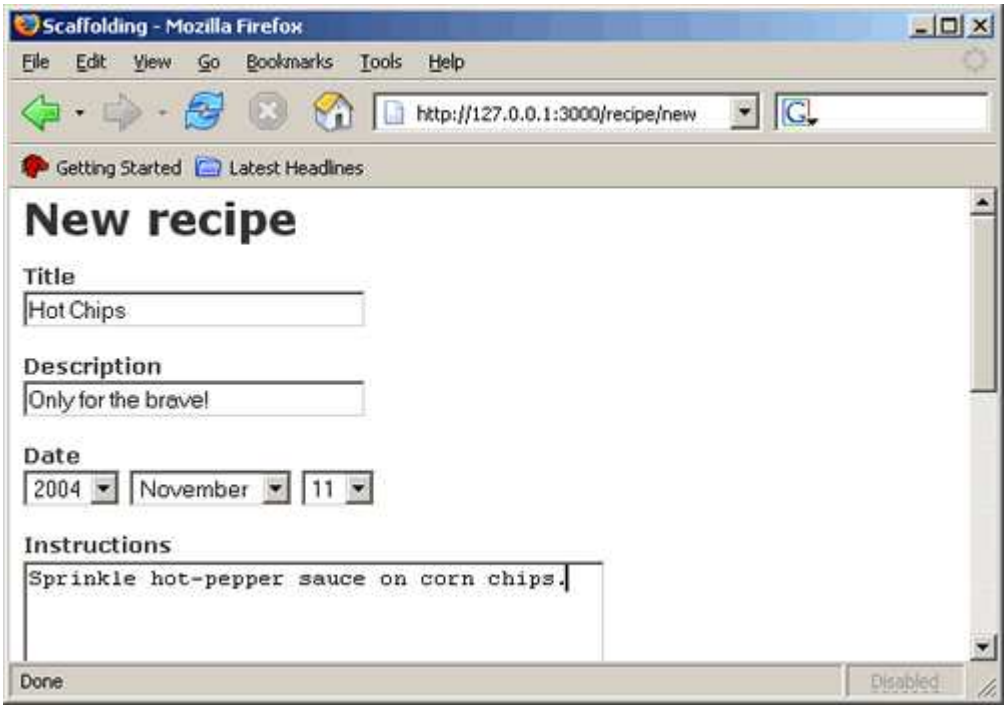


Figure 36. A new recipe

You should see the results, as in Figure 37.

Figure 37. A listing of all recipes

Add another one by clicking the "New recipe" link and entering the data, as in Figure 38.

Figure 38. Another new recipe

After you click Create you should see something like Figure 39.

Figure 39. A fuller list of all recipes

We now have an amazing amount of functionality, by merely building a database table and typing in a single line of code. It may not be pretty yet, but we'll fix that soon enough.

In the meantime, play around with adding, deleting, and editing recipes. Go ahead; I'll wait for you in the next section.

What Just Happened?

A single line of code, `scaffold :recipe`, brought everything to life. It let us begin working with our data model. With virtually no work on our part, it created the actions `list`, `show`, `edit`, and `delete`. It also created default view templates for each of these actions.

Of course, these actions and views are very plain--not the sort of thing you'd want users to see (unless they are total geeks).

The good news is that we can leave the scaffolding in place and slowly, one at a time, provide our own versions of the actions and views. Each time you create one of the actions or views it will override the scaffold's version. When you're done, simply remove the `scaffold` statement from the controller.

Before we do that, did you notice the URLs as you were playing around with your new cookbook? Rails tries very hard to present the user with pretty URLs. Rails URLs are simple and straightforward, not long and cryptic.

Creating Actions and Views

The page that shows the list of all recipes desperately needs improvement. The way to do that is to take over the handling of the `list` action from the scaffolding.

Edit *recipe_controller.rb* and add a `list` method similar to Figure 40.

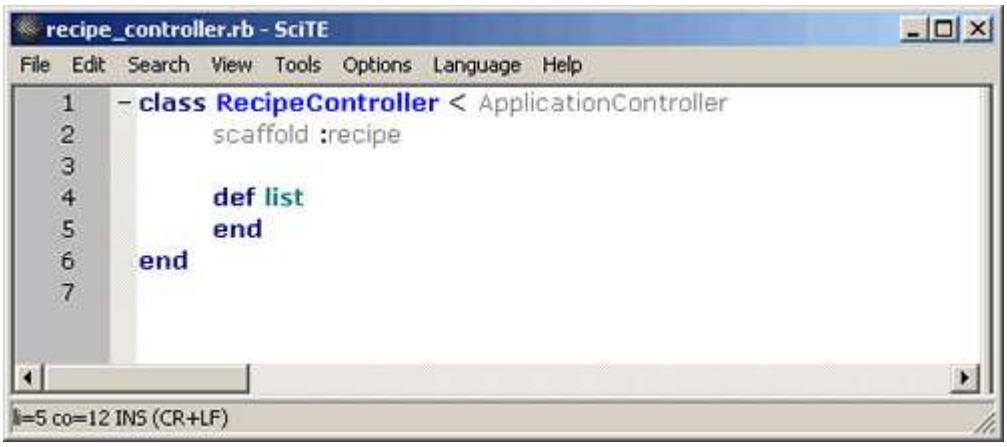


Figure 40. A new `list` method

Browse to `http://127.0.0.1:3000/recipe/list` and you should see something like Figure 41.

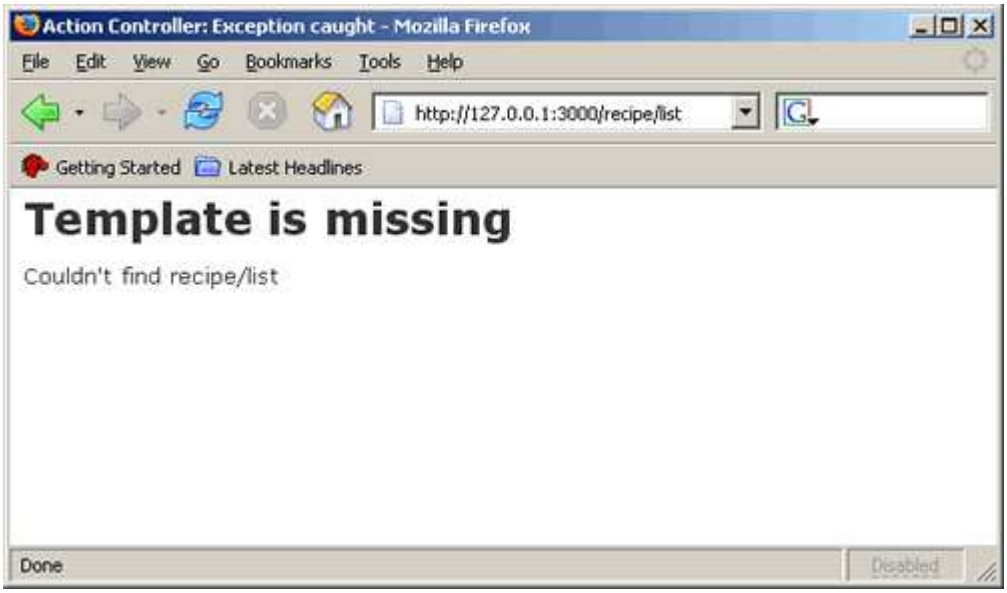


Figure 41. The results of the new `list` method

Because we just created our own definition for the `list` action, Rails no longer uses the scaffold version. Rails called our `list` method and then tried to find a view template to render. Because we did not create one, we received this "template missing" error. Let's create our own view template for the `list` action that only shows each recipe's title and date.

When we created our recipe controller, the `generate controller` script also created a view directory where we can place the HTML templates that the recipe controller can display. We need to create a template file named *list.rhtml* in *c:\rails\cookbook\app\views\recipe*. If you have worked with JSP or ASP pages, this will look familiar. It is simply an html file with Ruby code embedded within `<% %>` and `<%= %>` tags.

In the directory *c:\rails\cookbook\app\views\recipe*, create a file named *list.rhtml* containing the following:

```
<html>
<head>
<title>All Recipes</title>
</head>
<body>

<h1>Online Cookbook - All Recipes</h1>
<table border="1">
 <tr>
  <td width="80%"><p align="center"><i><b>Recipe</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 %></td>
   <td><%= recipe.date %></td>
  </tr>
 <% end %>
</table>
<p><%= link_to "Create new recipe", :action => "new" %></p>

</body>
</html>
```

Edit *recipe_controller.rb* and add the single line of code shown in Figure 42 to the `list` method.

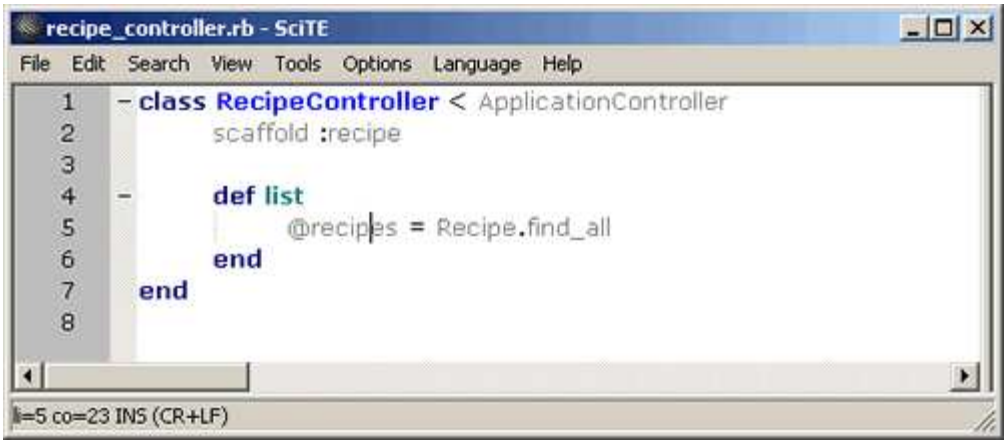


Figure 42. Listing all recipes

Refresh your browser and you should see something like Figure 43.

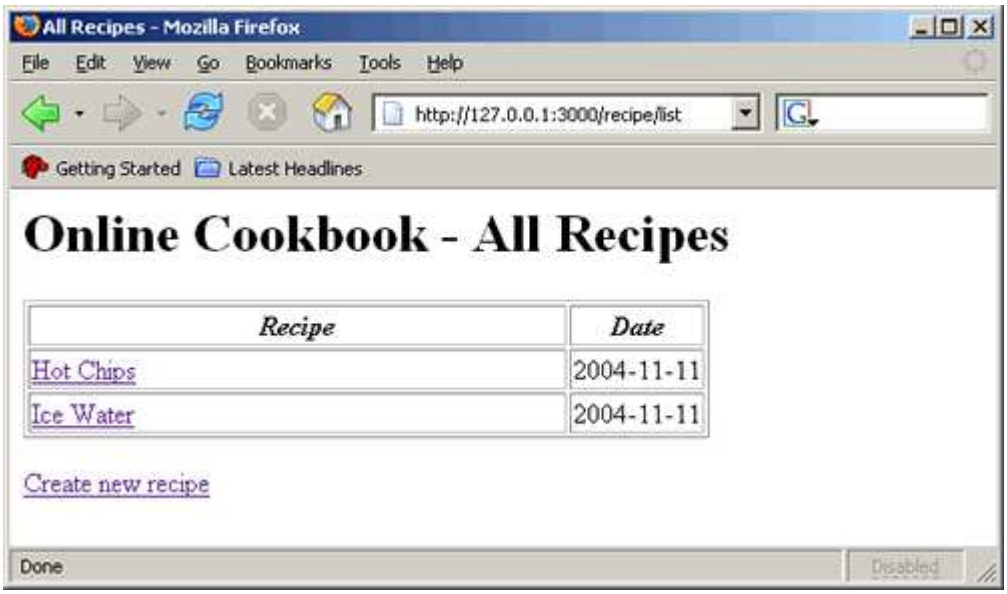


Figure 43. A nicer recipe list

Now this definitely looks better! How does it work?

```
def list
```

```
  @recipes = Recipe.find_all
end
```

When a user browses to http://127.0.0.1:3000/recipe/list, Rails will call the new `list` method we just created. The single line of code in the method asks the `Recipe` class for a collection of all recipes from the database, assigning the collection to the instance variable `@recipes`.

Next, Rails will look for a template to render and return to the browser. Most of our list template is standard HTML. The real action is in this section of the template:

```
<% @recipes.each do |recipe| %>
  <tr>
   <td><%= link_to recipe.title, :action => "show", :id => recipe.id %></td>
   <td><%= recipe.date %></td>
  </tr>
 <% end %>
```

This embedded Ruby code iterates through the collection of recipes retrieved in the controller. The first cell of the table row creates a link to the recipe's `show` page. Notice the attributes used on the recipe object (`title`, `id`, and `date`). These came directly from the column names in the `recipes` table.

Adding Categories to the Cookbook

We want to be able to assign a recipe to a category (like "dessert") and be able to list only those recipes that are in a particular category. To do this, we need to add a category table to the database, and a field to the recipe table specifying the category to which the recipe belongs.

In MySQL-Front, create a `categories` table. Remember to change the automatically created `Id` field to `id`, and then create a `name` field as a `varchar(50)`. The resulting table should look like Figure 44.

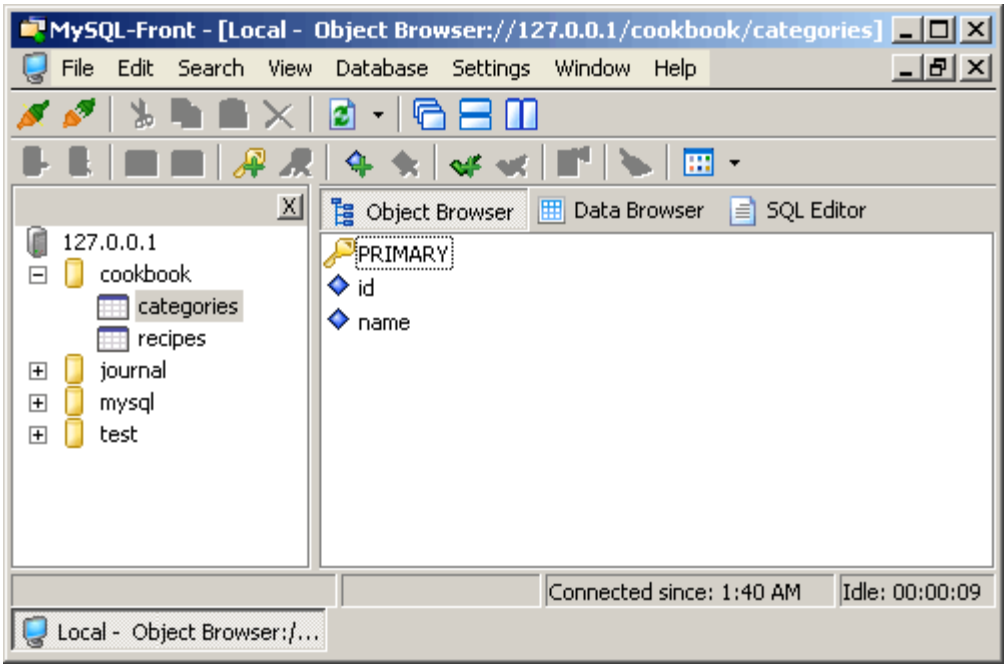


Figure 44. The `categories` table

We also need a category controller and a category model. Open a command window in the cookbook directory and run the commands (Figure 45):

```
ruby script\generate controller Category
ruby script\generate model Category
```

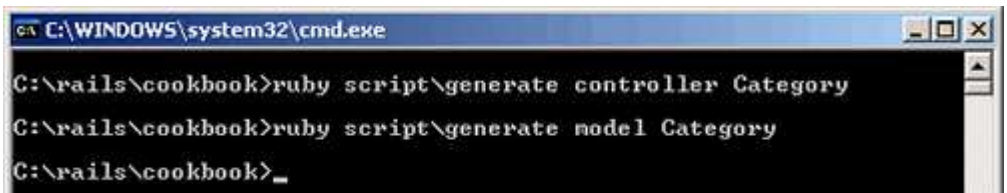


Figure 45. Creating the category model and controller

Finally, add scaffolding to the category controller. Edit *c:\rails\cookbook\app\controllers\category_controller.rb* and add the scaffolding shown in Figure 46.

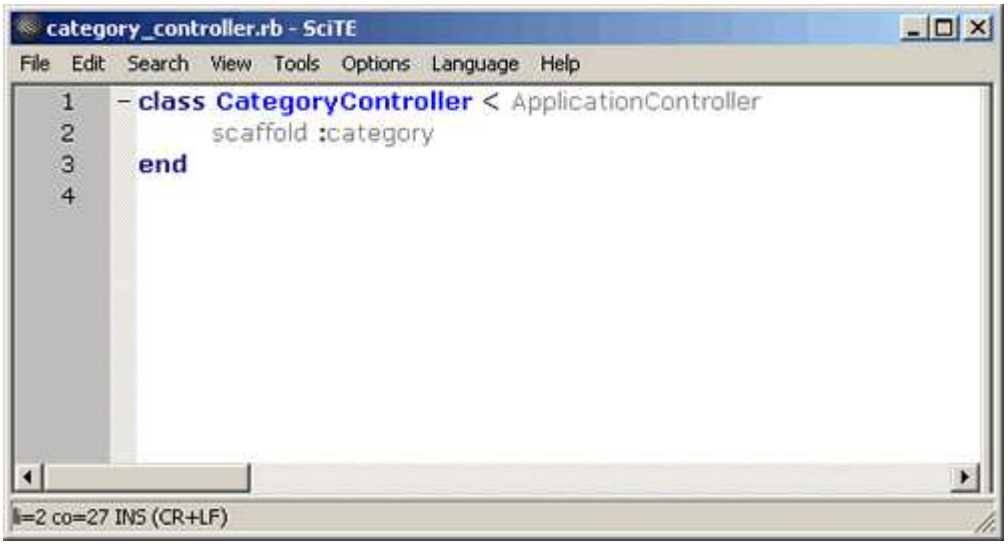


Figure 46. Category scaffolding

Browse to `http://127.0.0.1:3000/category/new` and create two categories named `Snacks` and `Beverages`. When you are done, you should see something like Figure 47.

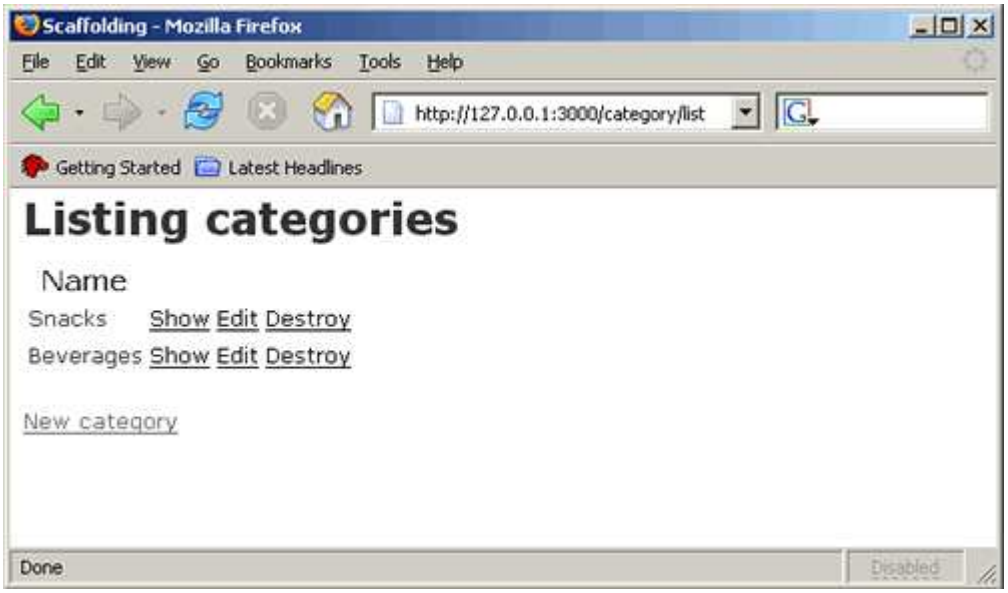


Figure 47. A listing of all categories

Assigning a Category to Each Recipe

The cookbook now has recipes and categories, but we still need to tie them together. We want to be able to assign a category to a recipe. To do this we need to add a field to our recipes table to hold the category `id` for each recipe, and we'll have to write an `edit` action for recipes that provides a drop-down list of categories.

First, add a `category_id` field to the `recipe` table as an `int(6)` to match the key of the `category` table. Figure 48 has the details.

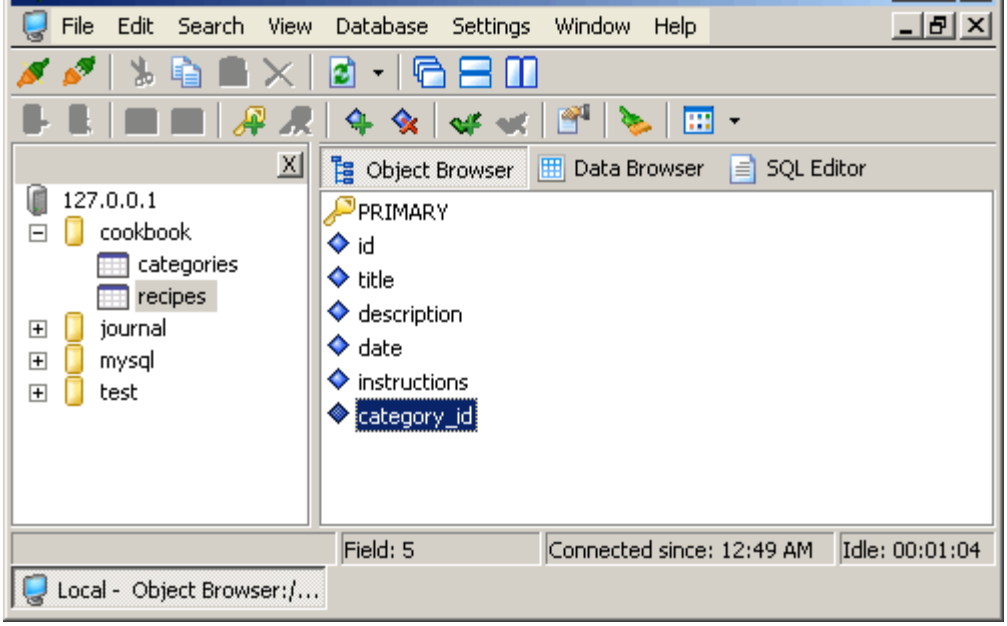


Figure 48. The `recipe` table with its new `category_id`

This will hold the `id` of the recipe's category. Now tell the `Recipe` model class about this too.

Edit *c:\rails\cookbook\app\models\recipe.rb* and *c:\rails\cookbook\app\models\category.rb* to add a single line to each model class, as shown in Figures 49 and 50:

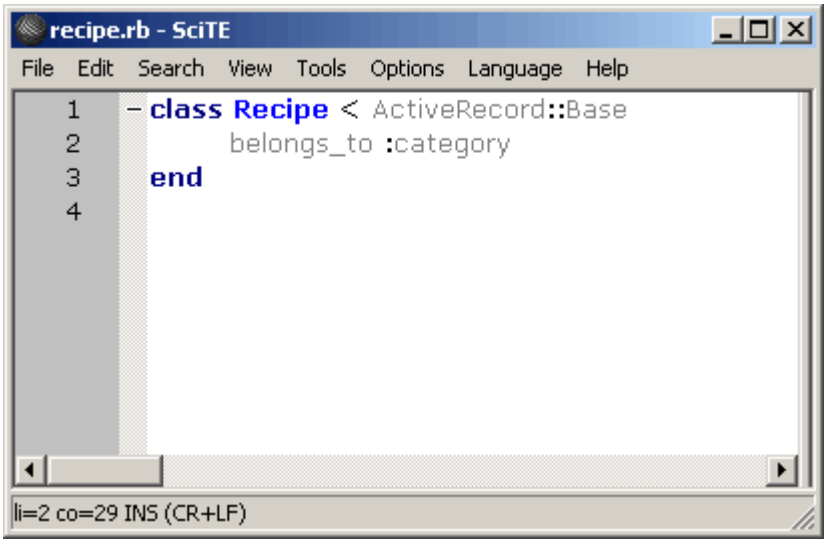


Figure 49. Setting relationships in the `Recipe` model

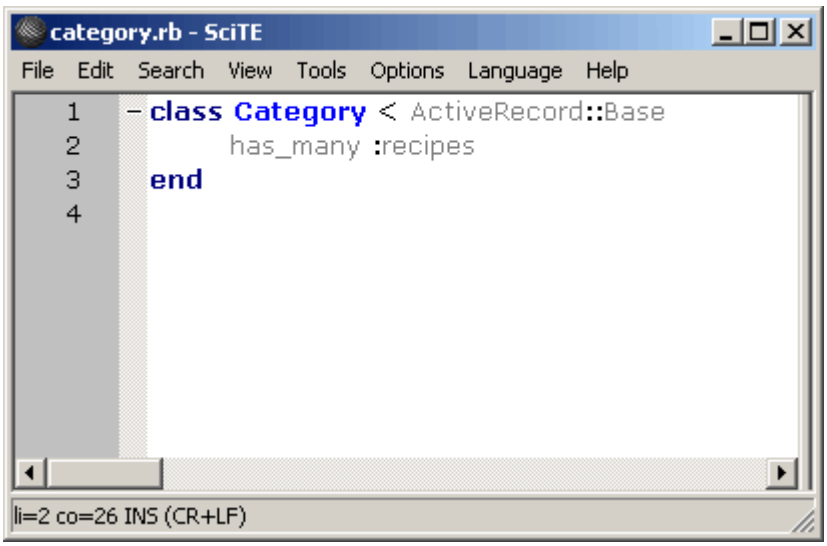


Figure 50. Setting relationships in the `Category` model

It should be pretty obvious that this tells Rails that a recipe belongs to a single category and that a category can have many recipes. These declarations actually generate methods to navigate these data relationships in Ruby code.

For example, if I have a recipe object in `@recipe`, I can find its category name with the code `@recipe.category.name`. Similarly, if I have a category object in `@category`, I can fetch a collection of all recipes in that category using the code `@category.recipes.`.

Now it's time to take over the `edit` recipe action and template from the scaffolding so that we can assign categories. Edit *c:\rails\cookbook\app\controllers\recipe_controller.rb* and add an `edit` method like in Figure 51.

Figure 51. The `Recipe` controller's new `edit` method

This creates two instance variables that the template will use to render the "edit recipe" page. `@recipe` is the recipe that we want to edit (the `id` parameter came in with the web request). `@categories` is a collection of all the categories in the database. The template will use it to create a drop-down list of category choices.

In the directory *c:\rails\cookbook\app\views\recipe*, create a file named *edit.rhtml* that contains the HTML template shown below. It's mostly standard HTML, with the main trick being the `<select>` and `<option>` tags that create the drop-down list of categories:

```
<html>
 <head>
  <title>Edit Recipe</title>
 </head>
 <body>
 <h1>Edit Recipe</h1>

 <form action="../update" method="POST">
  <input id="recipe_id" name="recipe[id]" size="30"
         type="hidden" value="<%= @recipe.id %>" />
  <p><b>Title</b><br>
  <input id="recipe_title" name="recipe[title]" size="30"
         type="text" value="<%= @recipe.title %>" />
  </p>
  <p><b>Description</b><br>
  <input id="recipe_description" name="recipe[description]"
         size="30" type="text"
         value="<%= @recipe.description %>" />
  </p>
  <p><b>Category:</b><br>

  <select name="recipe[category_id]">
   <% @categories.each do |category| %>
      <option value="<%= category.id %>"
        <%= ' selected' if category.id == @recipe.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">
    <%= @recipe.instructions %>
  </textarea> </p>
  <input type="submit" value="Update" />
 </form>

 <a href="/recipe/show/<%= @recipe.id %>">
   Show
 </a> |
  <a href="/recipe/list">
  Back
 </a>

 </body>
</html>
```

You can see the `@recipe` and `@categories` variables being used. Notice the section that loops through all of the categories to create a selection list. Look at the `<option>` tag and notice how it uses the current category assigned to the recipe being edited as the selected option. Study the template and then try it out.

Browse to `http://127.0.0.1:3000/recipe/list` and edit the recipe for "Ice Water." Change its category to "Beverages," as shown in Figure 52.

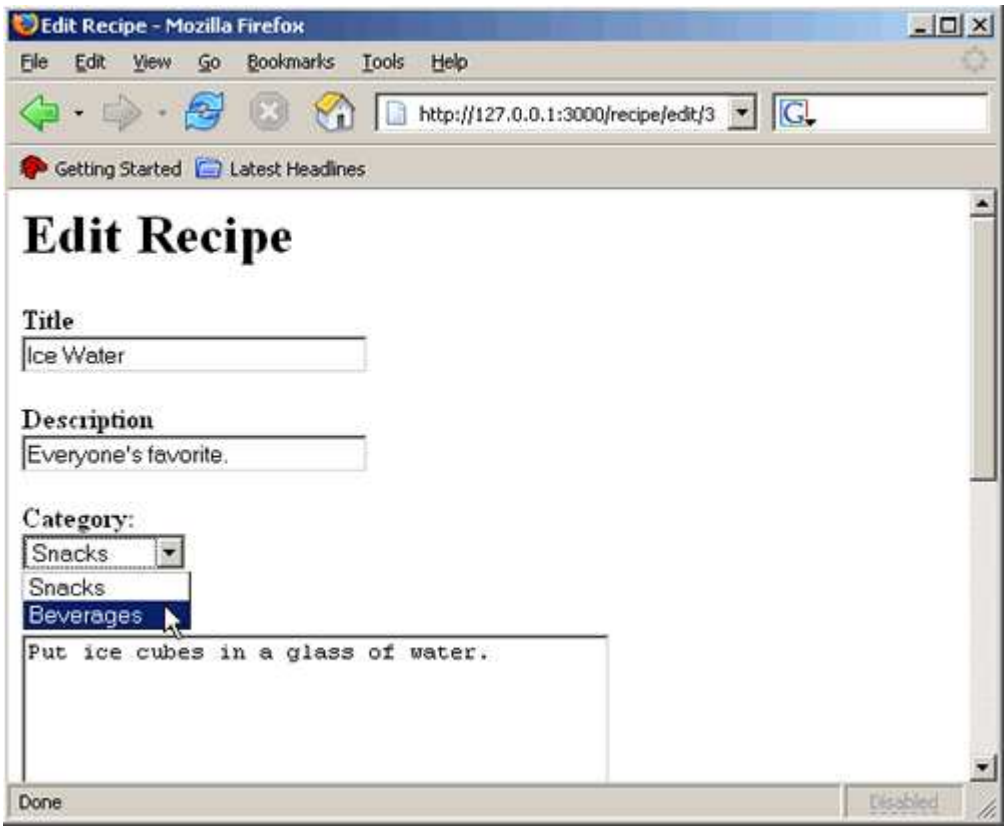


Figure 52. Changing the category for a recipe

Before moving on to the final step, make sure that all recipes in the database have a category. Edit each of them, select a category, and update them. If you don't do this, the next step will give you errors.

Displaying Categories in our List of All Recipes

This is the final step. Modify the list template that we made earlier to display each recipe's category.

Edit the file *c:\rails\cookbook\app\views\recipe\list.rhtml* to 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 %></td>
   <td><%= recipe.category.name %></td>
   <td><%= recipe.date %></td>
  </tr>
 <% end %>
</table>
<p><%= link_to "Create new recipe", :action => "new" %></p>

</body>
</html>
```

Now try it by browsing to http://127.0.0.1:3000/recipe/list. You should see something like Figure 53.

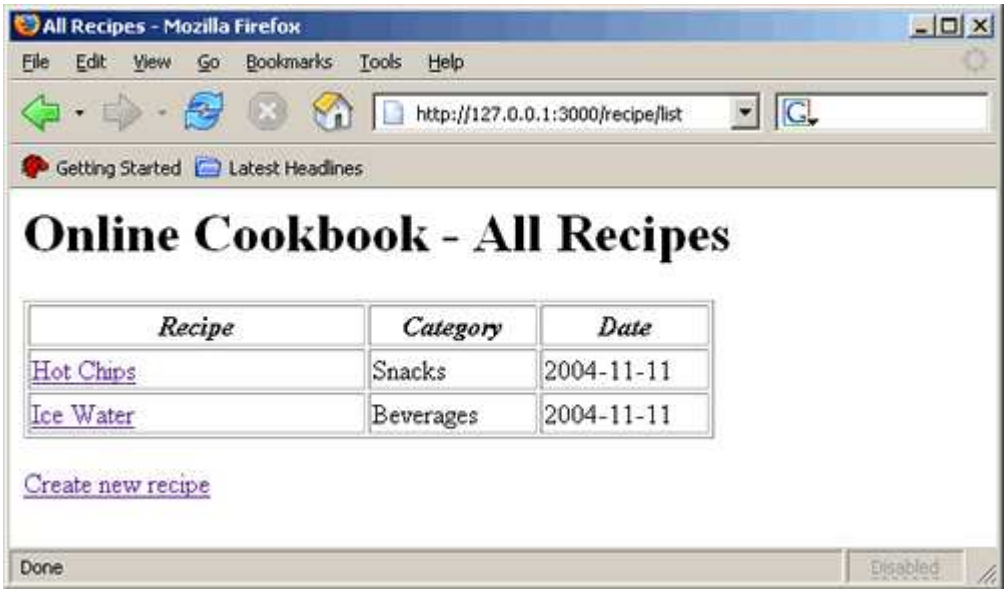


Figure 53. Recipes listed with categories

Exercises for the Reader

Congratulations, you've built a Rails web application! Of course, it still needs some work, but it is functional.

Here's some homework for you:

- 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 a list of 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.

This article is the first of a two-part series. Part two will implement the items listed above, but you don't have to wait for me--implementing them yourself could be a fun way to start on Rails development!

Related Reading

Parting Thoughts

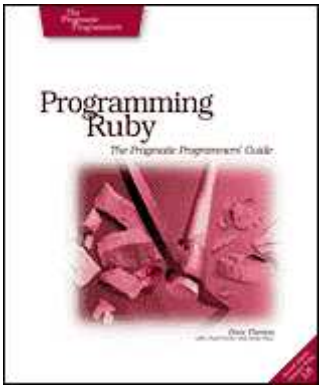
Ruby on Rails has taken web application development to a whole new level. You no longer

need to do the parts that used to be tedious work, because Rails does them for you. Even if you have to use a legacy database that does not use the Rails naming conventions, you don't have to give up the productivity advantages of using Rails--there is still a way to tell Rails explicitly what table and column names to use.

Now that you've seen firsthand how easy it can be to create a web application, why would you want to do it any other way?

Perhaps your employer has mandated a particular framework or language. You can still take a few days to prototype it in Rails, then go to your boss and say, "I've already finished writing our entire application in Ruby on Rails! If you prefer, we can still take the next few months to write it as we originally planned." <grin>

Editor's note: Want more Ruby on Rails? See Rolling with Ruby on Rails, part 2 and Ajax on Rails.



Programming Ruby
The Pragmatic Programmer's Guide, Second Edition
By Dave Thomas

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

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.