Ruby on Rails

OBJECTIVES

In this chapter you will learn:

- Basic Ruby programming.
- How to use the Rails framework.
- The Model-View-Controller paradigm.
- How to use `ActiveRecord` to model a database.
- How to construct web applications that interact with a database.
- How to create a web-based message forum.
- How to develop Ajax-enabled applications in Ruby on Rails.
- How to use the built-in Script.aculo.us library to add visual effects to your programs.

Convention is the ruler of all.
—Pindar

Where the telescope ends, the microscope begins. Which of the two has the grander view?
—Victor Hugo

. . . We grow more partial for the observer's sake.
—Alexander Pope

Those who cannot remember the past are condemned to repeat it.
—George Santayana

Let's look at the record.
—Alfred Emanuel Smith

All that matters is that the miraculous become the norm.
—Henry Miller

Outline

24.1 Introduction

Ruby on Rails (also known as **RoR** or just **Rails**) is a framework for developing data-driven web applications using the **Ruby** scripting language. A **web framework** is a set of libraries and useful tools that can be used to build dynamic web applications. Ruby on Rails is different from most other programming languages because it takes advantage of many conventions to reduce development time. If you follow these conventions, the Rails framework generates substantial functionality and perform many tasks for you. Ruby on Rails has built-in libraries for performing common web development tasks, such as interacting with a database, sending mass e-mails to clients or generating web services. In addition, Rails has built-in libraries that provide Ajax functionality (discussed in Chapter 15), to improve the user experience. Rails is quickly becoming a popular web development environment.

Ruby on Rails was created by David Heinemeier Hansson of the company 37Signals. After developing Basecamp, a web application written in Ruby that allows a business to organize multiple projects. Hansson extracted the reusable components to create the Rails framework. Since then, many developers have enhanced the Rails framework. For more information, visit our Ruby on Rails Resource Center at `www.deitel.com/RubyOnRails`. Full documentation of the Rails Framework can be found at `api.rubyonrails.org`.

24.2 Ruby

The first several examples are simple command-line programs that demonstrate fundamental Ruby programming concepts. The Ruby scripting language was developed by Yukihiro "Matz" Matsumoto in 1995 to be a flexible, object-oriented scripting language. Ruby's syntax and conventions are intuitive—they attempt to mimic the way a developer thinks. Ruby is an interpreted language.

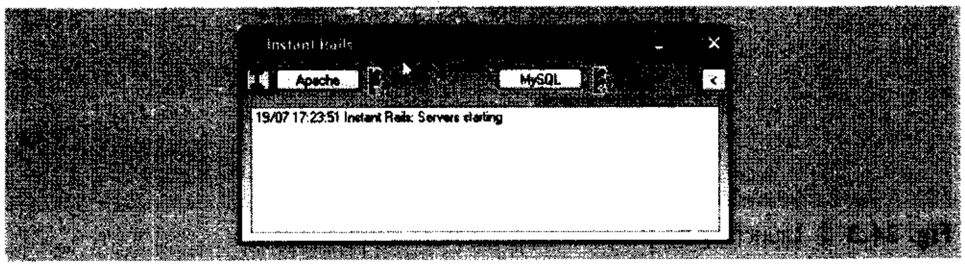
Installing Instant Rails

To run the Ruby scripts in this chapter, Ruby must first be installed on your system. In this chapter we use the **Instant Rails** package to run our applications. Instant Rails includes Ruby, Rails, MySQL, Apache, PHP and other components necessary to create and run Rails applications. PHP is used specifically for **phpMyAdmin**, a web interface to MySQL. Instant Rails is a stand-alone Rails development and testing environment.

To install Instant Rails, download Instant Rails 1.7 from `//rubyforge.org/frs/?group_id=904`. Once the zip file is downloaded, extract its contents to a folder on your hard drive.

After installing Instant Rails, make sure that you stop any existing web servers on your computer such as IIS or Apache— Instant Rails needs port 80 to be available for using **phpMyAdmin** to administer MySQL. If you are not using this tool then you don't need to stop other web servers on your computer. To run Instant Rails, navigate to the folder where you extracted the contents of the zip file and run **InstantRails.exe**. You should see a window similar to Fig. 24.1.

If you are using Mac OS X, there is an application similar to Instant Rails called Locomotive. You can download Locomotive from `locomotive.raaum.org`. Linux users might want to try LinRails (available from `linrails.thembid.com`). Another program useful for Rails development is Aptana Radrails—a free, open-source IDE. Radrails can be downloaded from `www.aptana.com/download_rails_rdt.php`.



```
19/07 17:23:51 Instant Rails: Servers starting
```

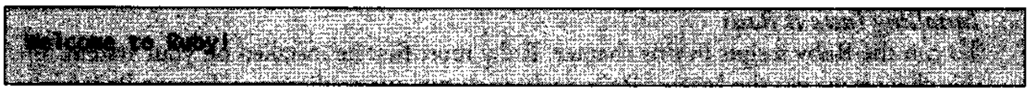
Fig. 24.1 | Instant Rails application running.

Printing a Line of Text

Figure 24.2 presents a simple Ruby program that prints the text `"Welcome to Ruby!"`. Lines 1–2 are single-line comments that instruct the interpreter to ignore everything on the current line following the # symbol. Line 3 uses the method **`puts`** that takes a single parameter (a string) and prints the text to the terminal, followed by a newline. A method can have parentheses surrounding its parameters, but this is not typical in Ruby unless they are used to avoid ambiguity. A line of Ruby code does not have to end with a semicolon, although one can be placed there. The `puts` method automatically adds a newline escape sequence (\n) at the end of the string if one is not explicitly added.

```
[illegible]
[illegible]
puts "Welcome to Ruby!"
```

Fig. 24.2 | Simple Ruby program. (Part 1 of 2.)



```
Welcome to Ruby!
```

Fig. 24.2 | Simple Ruby program. (Part 2 of 2.)

Running a Ruby Script

A Ruby script can be run several ways. One is to use the **Ruby interpreter.** To do so, launch Instant Rails, click the **I** button in the top-left corner and select **Rails Applications > Open Ruby Console Window** from the drop-down menu (see Fig. 24.3).

In the console, use the `cd` command to navigate to the directory where `welcome.rb` is located, then enter `ruby welcome.rb`. Figure 24.4 shows the Ruby interpreter executing the Ruby file from Fig. 24.2 in the **Ruby Console** window.

Ruby can also execute interactively, using **IRB (Interactive Ruby).** IRB interprets Ruby code statement by statement. This is useful for debugging code and for experimenting with Ruby functionality. IRB can be run through Instant Rails by typing `IRB` in the Ruby Console. Figure 24.5 shows simple Ruby statements interpreted in IRB.

The code after the prompt (`irb(main):001:0>`) shows the statement that was executed using the Ruby interpreter in Fig. 24.4. It sends the same string to the output, then returns the value of method `puts`, which is `nil`, an object that represents nothing in Ruby.

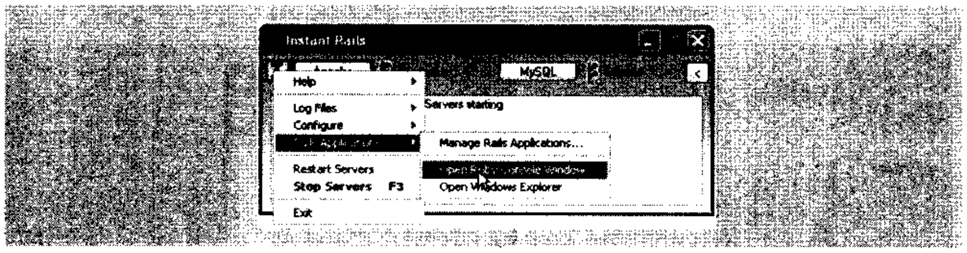
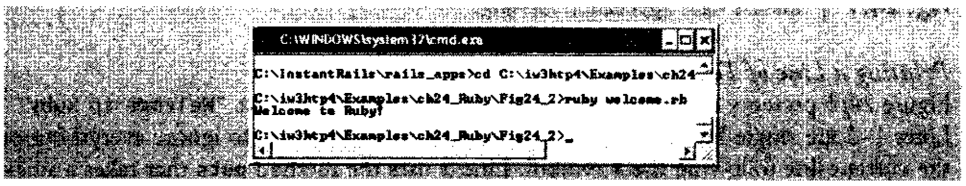


Fig. 24.3 | Launching the Ruby Interpreter in Instant Rails.



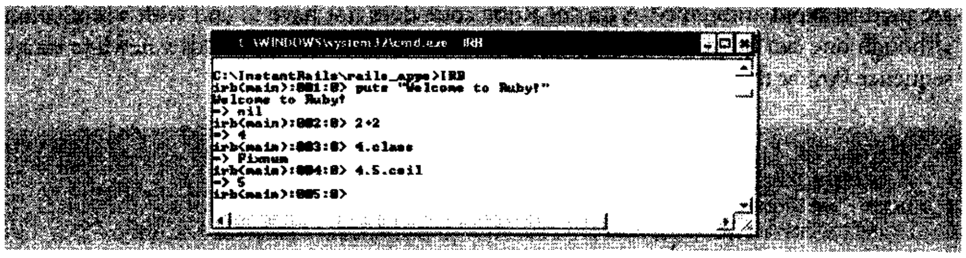
```
C:\WINDOWS\system32\cmd.exe

C:\InstantRails\rails_apps>cd C:\iw3htp4\Examples\ch24

C:\iw3htp4\Examples\ch24_Ruby\Fig24_2>ruby welcome.rb
Welcome to Ruby!

C:\iw3htp4\Examples\ch24_Ruby\Fig24_2>_
```

Fig. 24.4 | Using the Ruby interpreter to run a simple Ruby script.



```
C:\WINDOWS\system32\cmd.exe - IRB

C:\InstantRails\rails_apps>IRB
irb(main):001:0> puts "Welcome to Ruby!"
Welcome to Ruby!
=> nil
irb(main):002:0> 2+2
=> 4
irb(main):003:0> 4.class
=> Fixnum
irb(main):004:0> 4.5.ceil
=> 5
irb(main):005:0>
```

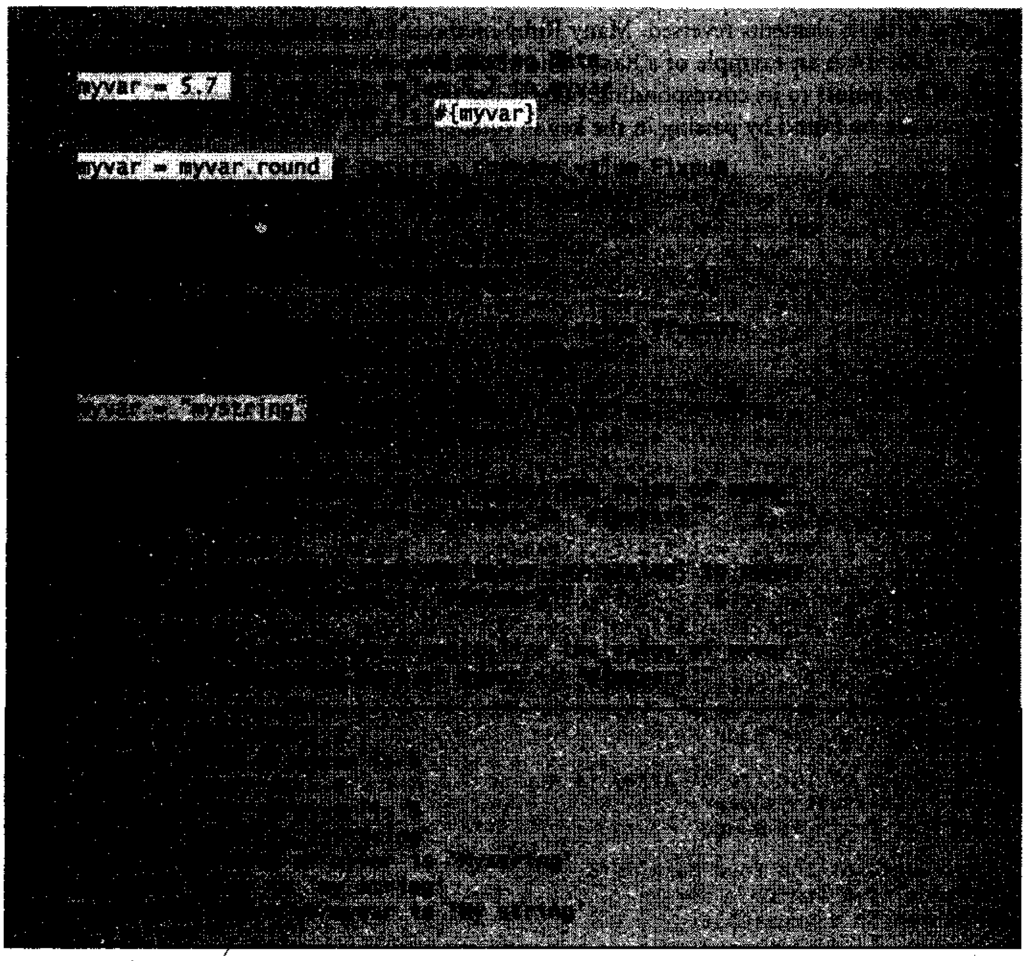
Fig. 24.5 | Using Interactive Ruby to execute Ruby statements.

The code after the second IRB prompt sends the arithmetic expression 2+2 to the interpreter, which evaluates the expression and returns 4. The code after the third prompt requests the class type of the number 4. The interpreter returns **Fixnum**—a class that represents integers in Ruby. Last, the code after the fourth prompt calls the ceil method of 4.5 to round the number up to the next whole-number value. The IRB returns 5. Type exit to quit IRB.

Variables and Data Types in Ruby

Like most scripting languages, Ruby uses **dynamic typing**, which allows changes to a variable's type during runtime. There are several variable types in Ruby, including String and Fixnum. Everything is an object in Ruby, so you can call methods on any piece of data. Figure 24.6 invokes methods on numeric and string data.

Line 3 initializes the variable myvar. Setting myvar to 7.5 temporarily makes it a Float object. The highlighted portion of line 4 is an example of **interpolation** in Ruby. It inserts the object value of the variable inside the braces into the string. Lines 6 and 12 invoke the



```
[illegible]
myvar = 5.7
[illegible] #{myvar}
[illegible]
myvar = myvar.round [illegible]
[illegible]
```

Fig. 24.6 | Method calls on numeric and string data.

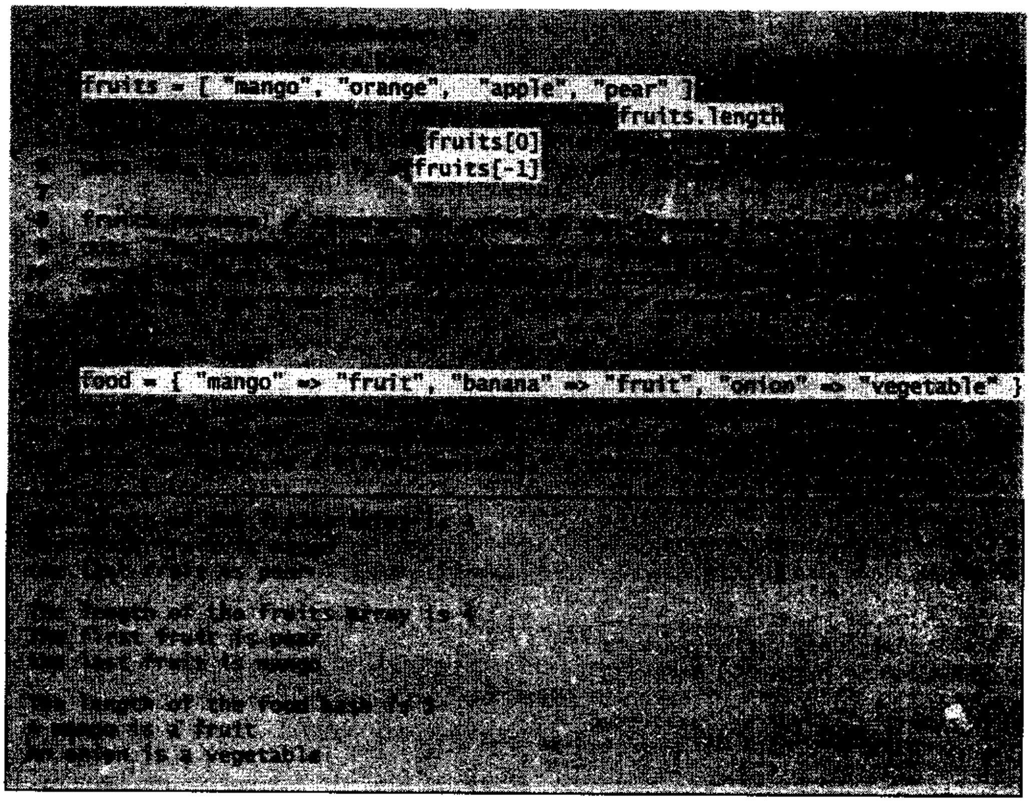
round method on the Float object to demonstrate rounding a value up or down, respectively. The type of myvar changes to String in line 15. Lines 18 and 24 changes the first letter of the first word of this String by calling its capitalize method. A list of the available methods for the Ruby types can be found at www.ruby-doc.org/core/.

*Using **Arrays** and **Hashes***

Ruby provides both **Arrays** and **Hashes** to store data. Each stores a list of objects. In an Array, indices of type Fixnum are used to select an Object from the Array. In a Hash, Objects are mapped to other Objects in key/value pairs. Figure 24.7 shows an example of using both an Array and a Hash to store information.

Line 3 instantiates a Ruby Array. Array elements can be accessed by their index number in square brackets (line 5). You may also traverse Arrays backward by using negative number indices. For example line 6 outputs the last array element. Line 8 reverses the elements in the Array with method **reverse!**. The exclamation point after the method name is a Ruby convention indicating that the object on which the method is called will be modified. Method reverse without an exclamation point returns a copy of the original array with its elements reversed. Many Ruby methods follow this convention.

Line 14 is an example of a Hash. The key/value pairs are separated by commas, and each key points to its corresponding value using the => operator. The value of a hash element can be found by passing in the key in square brackets, as shown in lines 16–17.



```
fruits = [ "mango", "orange", "apple", "pear" ]
[illegible] fruits.length
[illegible] fruits[0]
[illegible] fruits[-1]

fruits[illegible]
[illegible]

food = { "mango" => "fruit", "banana" => "fruit", "onion" => "vegetable" }
[illegible]
```

```
The length of the fruits array is 4
The first fruit is pear
The last fruit is mango
The length of the food hash is 3
A mango is a fruit
An onion is a vegetable
```

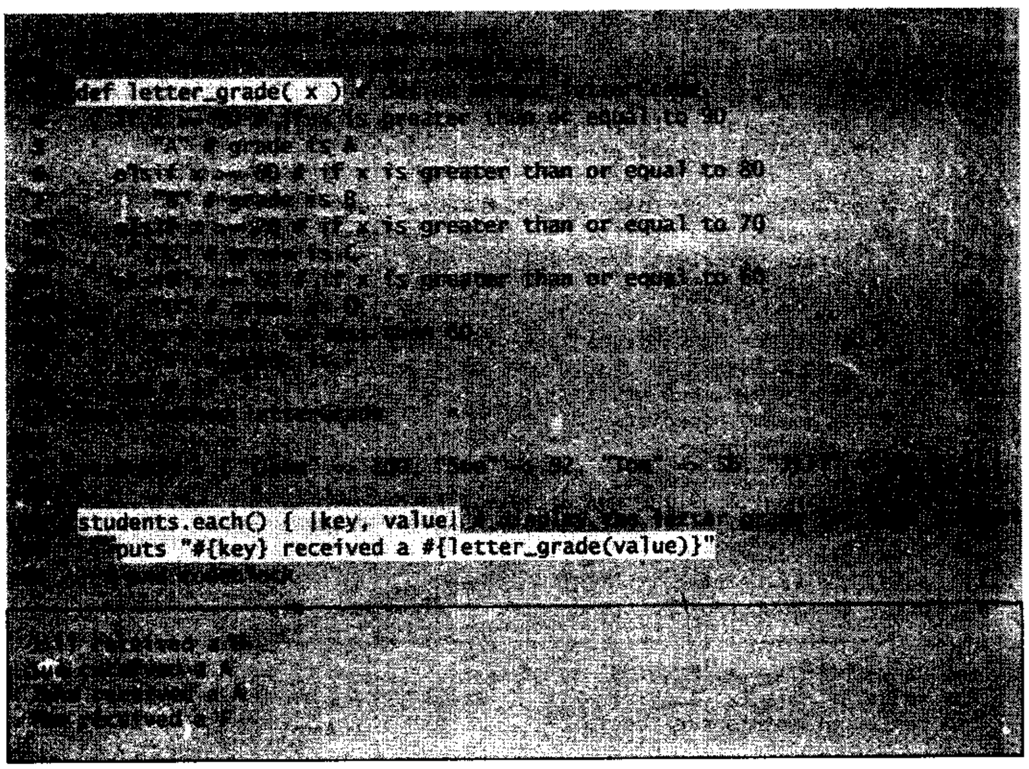
Fig. 24.7 | Arrays and hashes in Ruby.

Conditionals, Loops and Code Blocks

Like any other programming language, Ruby provides selection and repetition statements. In addition, Ruby has support for code blocks—groupings of Ruby statements that can be passed to a method as an argument. Figure 24.8 shows a program that returns a student's letter grade based on a numerical grade.

Lines 3–15 of Fig. 24.8 contain a Ruby method definition. Methods must be defined in a program before they are used. All methods start with **def** and end with **end**. Methods do not have to specify parameter types, but they must specify the name of each parameter. Lines 4–14 show a nested `if...elsif...else` statement that returns an appropriate letter grade based on the numeric value the method receives as an argument. If a method does not include an explicit return statement Ruby returns the last value or variable it encounters when executing the function.

Line 17 defines a `Hash` of students and their numeric grades. Lines 19–21 show an example of a code block. A method may have a parameter containing a block of code, such as the `each` method. The block of code appears in brackets directly after the method call. A code block is similar to a method, in that parameters can be passed into it. The parameters for a code block are given between pipe characters (|) and are separated by commas. The parameters are followed immediately by the code block's statements. The code block in lines 19–21 outputs a line of text based on the key/value pair of every key in the `Hash`.



```
def letter_grade( x ) # [illegible]
   [illegible] is greater than or equal to 90
      "A" # grade is A
   elsif x >= 80 # if x is greater than or equal to 80
      "B" # grade is B
   [illegible] if x is greater than or equal to 70
      [illegible] is C
   [illegible] if x is greater than or equal to 60
      [illegible] D
   [illegible]
[illegible]

[illegible]

students.each() { |key, value| # [illegible]
   puts "#{key} received a #{letter_grade(value)}"
[illegible]
```

```
[illegible] received a [illegible]
[illegible] received a [illegible]
[illegible] received a A
[illegible] received a F
```

Fig. 24.8 | Conditionals, loops and codeblocks.

Classes

You can create your own classes and instantiate objects. Classes enable you to encapsulate methods and data. Figure 24.9 shows a class named Point that stores *x*-*y* coordinates.

Line 3 begins the class definition with the keyword class followed by the class name. The initialize method (lines 7–11), like constructors in other object-oriented languages, is used to declare and initialize an object's data. When each instance of a class maintains its own copy of a variable, the variable is known as an **instance variable**. Lines 8–9 use the @ symbol to define the instance variables x and y. Classes can also have **class variables** that are shared by all copies of a class. Class variables always begin with @@ (line 4) and are visible to all instances of the class in which they are defined. Line 10 increments @@num_points every time a new Point is defined.

You can create new classes by inheriting from existing ones and providing your own additional or enhanced functionality. Lines 14–16 override the inherited **to_s method**, which is a method of all Ruby objects. When an object is concatenated with a string, the to_s method is implicitly called to convert the object to its string representation. Class Point's to_s method for the Point class returns a string containing the *x*-*y* coordinates.

```
class Point
  @@num_points = 0

  def initialize(x, y)
    [illegible]
     @@num_points +=1
  [illegible]

  def to_s
    [illegible]

p = Point.new( 8, 9 )
q = Point.new( 1, 1 )
[illegible]
```

```
[illegible]
[illegible]
[illegible]
```

Fig. 24.9 | A Ruby class.

24.3 Rails Framework

While users have benefitted from the rise of database-driven web applications, web developers have had to implement rich functionality with technology that was not designed for this purpose. The Rails framework combines the simplicity of Ruby with the ability to rapidly develop database-driven web applications.

Model-View-Controller

Ruby on Rails is built on the philosophies of **Convention over Configuration** and **Don't Repeat Yourself** (**DRY**). If you follow certain programming idioms, your applications will require minimal configuration, and Rails will generate substantial portions of your web applications for you. One of these conventions is using the **Model-View-Controller** (**MVC**) design pattern, which splits the application into the business logic aspects handled by the model and the design aspects handled by the view. The controller handles client requests by obtaining information from the model and rendering it to the view.

The MVC architectural pattern separates application data (contained in the **model**) from graphical presentation components (the **view**) and input-processing logic (the **controller**). Figure 24.10 shows the relationships between components in MVC.

The controller implements logic for processing user input. The model contains application data, and the view presents the data from the model. When a user provides input, the controller modifies the model with the given input. When the model changes, the controller notifies the view so that it can update its presentation with the changed data.

MVC does not restrict an application to a single view and a single controller. In a more sophisticated program, there might be two views of a document model. One view might display an outline of the document and the other might display the complete document. An application also might implement multiple controllers—one for handling keyboard input and another for handling mouse selections. If either controller makes a change in the model, both the outline view and the print-preview window will show the change immediately when the controller notifies all views of changes.

The primary benefit to the MVC architectural pattern is that developers can modify each component individually without having to modify the others. For example, developers could modify the view that displays the document outline without having to modify either the model or other views or controllers.

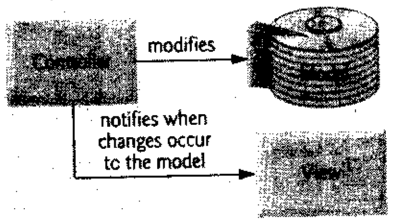


Fig. 24.10 | Model-View-Controller architecture.

Overview

In the following examples, we show how to create a Ruby on Rails application. We show how a controller can be used to send information to the client directly, and how a controller can render a view for a cleaner and more organized design. We then show how to set up a database in a Ruby on Rails application. Finally, we show how to generate a model to be the front end of a database in a dynamic web application.

Creating a Rails Application

The Instant Rails package comes with a full install of Rails that includes `ActiveRecord`, `ActionView`, and `ActionController`. **`ActiveRecord`** is used to map a database table to an Object. **`ActionView`** is a set of helper methods to modify user interfaces. **`ActionController`** is a set of helper methods to create controllers. To generate an empty Rails application in Instant Rails, click the I button and select **Rails Applications > Manage Rails Applications...** from the drop-down menu to display the **Rails Applications** window. In that window click the **Create New Rails App...** button. In the console that appears, type `rails` *Application Name* at the command line to create a directory named *Application Name* with a prebuilt directory structure inside. For the first example, use `Welcome` as the application name. Figure 24.11 shows the directory structure that is automatically generated by Rails. The directories that we'll be primarily concerned with are `app\controllers`, `app\models`, and `app\views`.

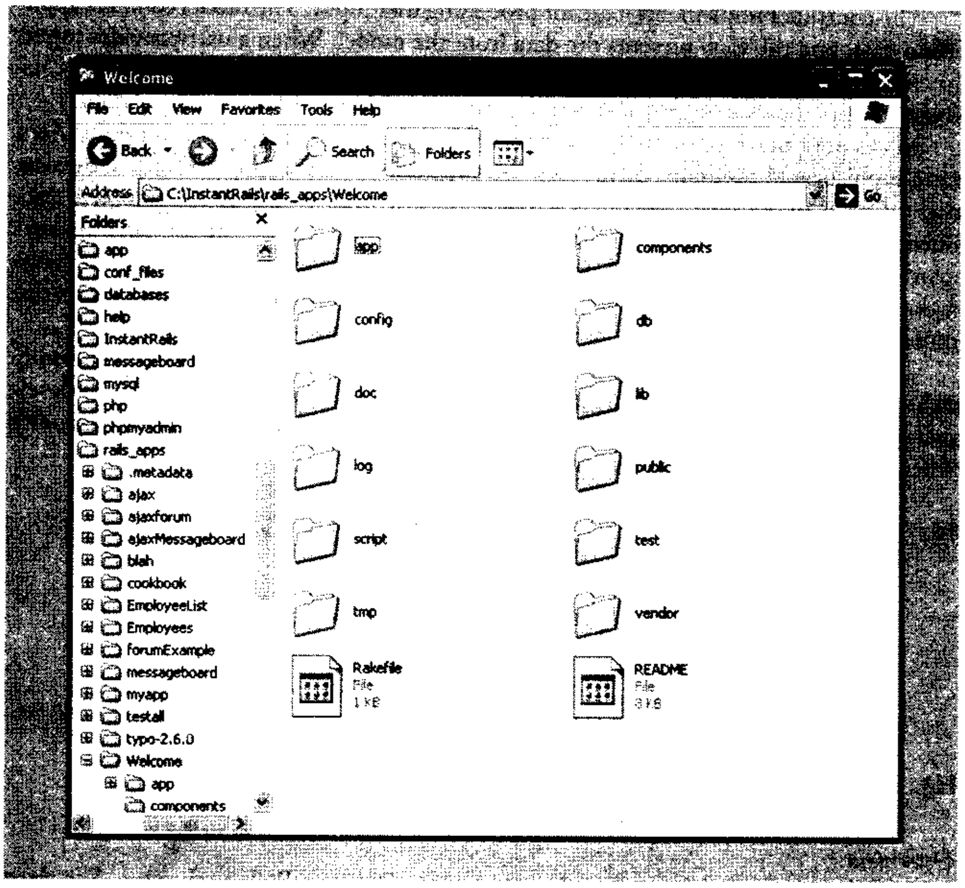


Fig. 24.11 | Rails directory structure for a new Rails application.

24.4 ActionController and ActionView

Ruby on Rails has two classes that work together to process a client request and render a view. These classes are `ActionController` and `ActionView`.

Rails Controller

To generate a controller in Rails, you can use the built-in `Controller generator`. To do that, open the **Ruby Console** window and navigate to the application directory by typing in:

```
cd pathToInstantRails\rails_apps\applicationName
```

To generate a controller for the welcome application type:

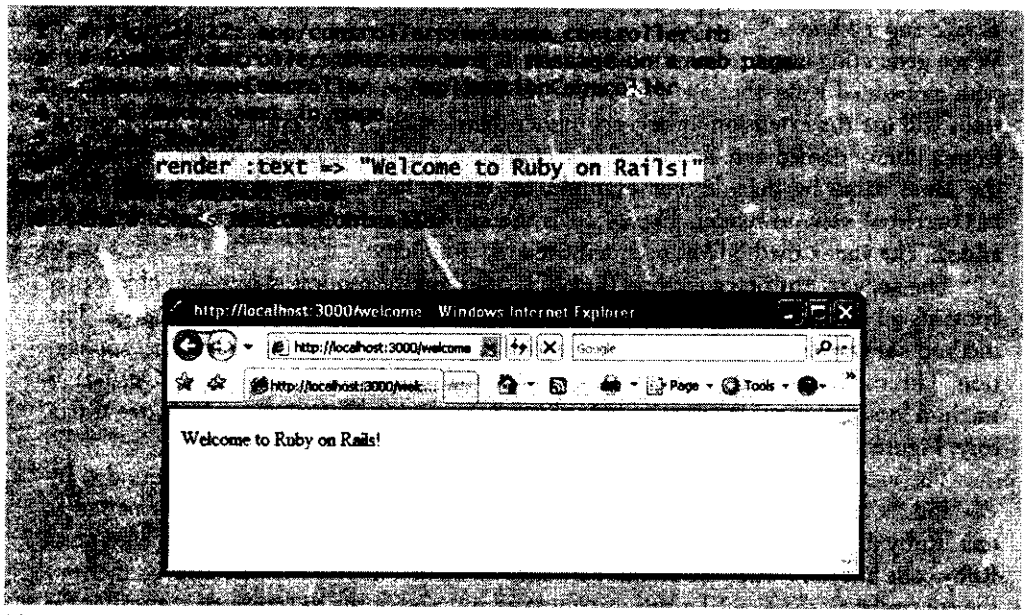
```
ruby script/generate controller Welcome
```

This creates several files including `welcome_controller.rb`, which contains a class named `WelcomeController`. Figure 24.12 shows a controller for our Welcome example containing only one method.

Line 3 defines a class `WelcomeController` that inherits from `ApplicationController`. `ApplicationController` inherits from `ActionController::Base`, which provides all of the default functionality for a controller. The method in lines 5–7 renders text in XHTML format to the browser using the `render` method

Line 6 specifies the `text` parameter of the render variable using a **ruby symbol**. Symbols are identifiers preceded by a colon that have a particular value or variable associated with them. When specifying a parameter in a method the notation is as follows:

parameter_symbol => parameter_value



```
[illegible] app/controllers/welcome_controller.rb
[illegible] message on a web page.
[illegible] ApplicationController
[illegible]

    render :text => "Welcome to Ruby on Rails!"
[illegible]
```

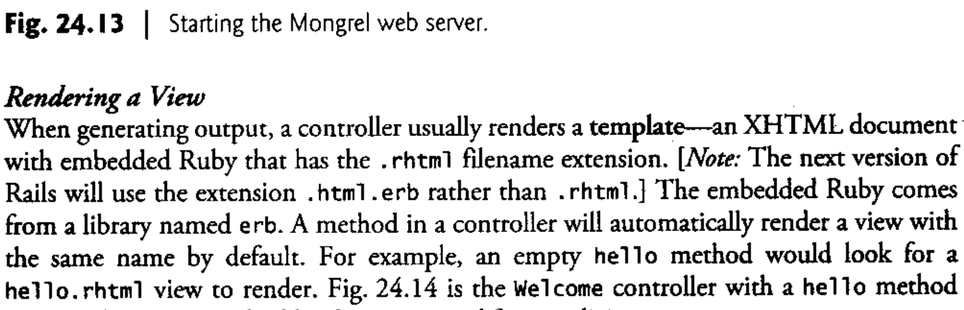
http://localhost:3000/welcome - Windows Internet Explorer

Welcome to Ruby on Rails!

Fig. 24.12 | Simple controller that renders a message on the web page.

Running Ruby on Rails
A Ruby on Rails application must be run from a web server. In addition to Apache, Instant Rails comes with a built-in web server named Mongrel, which is easy to use to test Rails applications on the local machine. You can start the Mongrel server through Instant Rails by going to the **Rails Application** window, selecting the **Welcome** application from the list and clicking the **Start with Mongrel** button (Fig. 24.13).

One important feature of Rails is its URL mapping ability. Rails automatically sets up your web application in a tree structure, where the controller's name in lowercase is the directory, and the method name is the subdirectory. Since the controller name is `Welcome` and the method name is `index`, the URL to display the text in Figure 24.12 is `http://localhost:3000/welcome/index`. Notice in the screen capture of Figure 24.12 that the URL is simply `http://localhost:3000/welcome`. The default action called on any controller is the one specified by the method `index`. So, you do not need to explicitly invoke the `index` in the URL to render the text in line 6.



```
** Starting Mongrel listening at 0.0.0.0:3000
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready.  INT => stop (no restart).
** Mongrel available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Fig. 24.13 | Starting the Mongrel web server.

Rendering a View
When generating output, a controller usually renders a **template**—an XHTML document with embedded Ruby that has the `.rhtml` filename extension. [*Note:* The next version of Rails will use the extension `.html.erb` rather than `.rhtml`.] The embedded Ruby comes from a library named `erb`. A method in a controller will automatically render a view with the same name by default. For example, an empty `hello` method would look for a `hello.rhtml` view to render. Fig. 24.14 is the `Welcome` controller with a `hello` method added. The `index` method has been removed for simplicity.

The `server_software` method (line 6) is called on the **request object**—an object that contains all of the environment variables and other information for that web page. The method **`server_software`** returns the name of the server that is running the web application. This name is stored in an instance variable that will be used by the view. Our `hello` method looks for a `hello.rhtml` file in the web application's `app/views/welcome` directory. Figure 24.15 shows a sample `hello.rhtml` file.

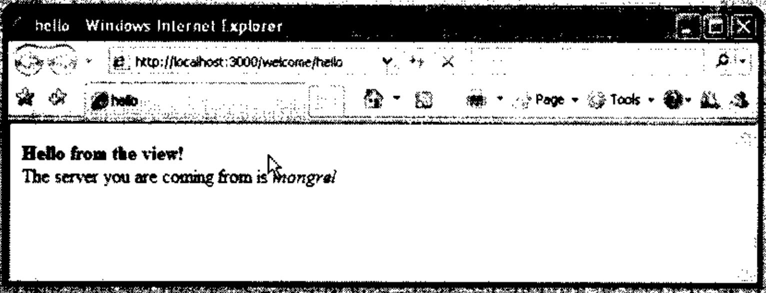
The view consists mostly of XHTML. The `erb` is shown in line 14, surrounded by `<%=` and `%>` tags. Everything between these tags is parsed as Ruby code and formatted as text. Ruby delimiters without an equals sign—`<% %>`—represents statements to execute as Ruby code but not formatted as text. The `@server_name` variable is passed in directly from the controller in the view. To run this application, modify the `welcome.rb` controller file to look like Figure 24.14. Then go to the `/app/views/welcome` directory, create the hello.rhtml file in Fig. 24.15. Run the welcome application on the Mongrel server (if it is not already running) and direct your browser to the URL http://localhost:3000/welcome/hello.

```
# Fig. 24.14: app/controllers/welcome_controller.rb
# Simple controller that passes a parameter to the view.
class WelcomeController < ApplicationController
   # set server_name to server information
   def hello
      @server_name = request.server_software # retrieve software of server
   end # method hello
end # class WelcomeController
```

Fig. 24.14 | Simple controller that passes a parameter to the view.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml11/DTD/xhtml11-strict.dtd">

<!-- Fig. 24.15: app/views/welcome/hello.rhtml -->
<!-- View that displays the server name. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <title>hello</title>
</head>
<body style = "background-color: lightyellow">
   <strong>Hello from the view!</strong><br />
   The server you are coming from is
   <em><%= @server_name %></em>
</body>
</html>
```


hello - Windows Internet Explorer
http://localhost:3000/welcome/hello
hello
Page
Tools

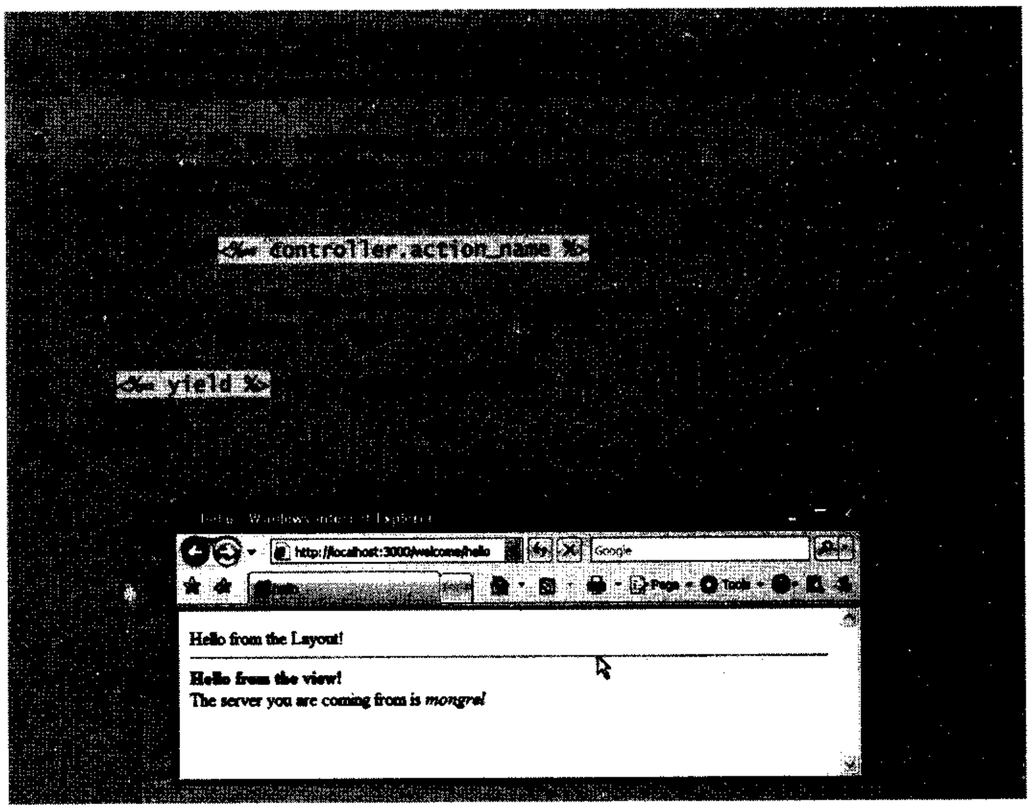
Hello from the view!
The server you are coming from is *Mongrel*

Fig. 24.15 | View that displays the name of the server.

Using a Layout

Often, information spans multiple web pages that could be viewed as a header or footer. Rails allows you to add headers and footers with a **layout**—a master view that is displayed by every method in a controller. A layout can refer to a template of the method that is being called, using **yield**. A layout has the same name as the controller, and is placed in the app/views/layouts directory. Figure 24.16 is a layout for the Welcome controller. To add a layout to the application create a welcome.rhtml file in the apps/views/layouts directory. To run this application, re-load the page from Fig. 24.15.

Line 9 invokes the **action_name** method on the controller object. This displays the name of the method that is currently being called in the controller. Instance variables defined in the controller are copied to both the layout and the view that the layout renders. Line 14 is a placeholder for the view content (hello.rhtml in this example) that is specific to the action called in the controller.



```
<%= controller.action_name %>

<%= yield %>
```

Hello from the Layout!

Hello from the view!
The server you are coming from is *mongrel*

Fig. 24.16 | Layout that displays a greeting.

24.5 A Database-Driven Web Application

The third tier of a typical Rails application—the model—manages the data used in the app lication. In this section, we set up a database and build a fully functional web application using the ActionView and ActionController classes that we introduced in Section 24.4. We create an application that allows the user to browse and edit an employee list. To create this application's structure, type rails Employees in the **Ruby Console** window.

Object Relational Mapping

Rails makes extensive use of Object-Relational Mapping (ORM) in its web framework. ORM maps a table to application objects. The objects that Rails uses to encapsulate a database inherit from **ActiveRecord**. By using ActiveRecord and Rails conventions, you can avoid a lot of explicit configuration.

One ActiveRecord convention is that every model that extends ActiveRecord::Base in an application represents a table in a database. The table that the model represents is, by convention, the lowercase, pluralized form of the model. For example, if there were a messages table in your database, Message would be the name of the model representing it. ActiveRecord follows many standard English pluralization rules as well, which means that a Person model would automatically correspond to a people table. Furthermore, if a people table has a first_name column, the Person model would have a method named first_name that returns the value in that column. ActiveRecord does this for you with no additional configuration.

Creating the Database

Before creating a model using ActiveRecord, we need to create the database it will use. You can do that using MySQL's mysqladmin command. Rails will automatically look for a database with the name *applicationName*_development to use as the development database. To create a database for the Employees application, launch the **Ruby Console** and type in mysqladmin -u root create employees_development. If no error is returned, the database was created successfully in the mysql/data directory of your InstantRails installation.

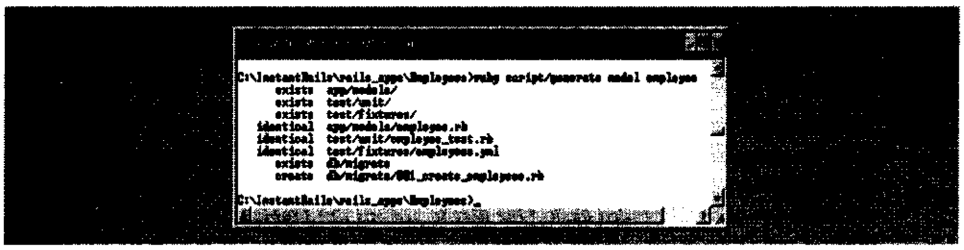
By default MySQL has the user name root and no password. If your settings are different you can modify the appropriate fields database.yml, located in the config folder in your application directory.

Creating the Employee Model

Since Rails separates the model from the rest of the application, we simply need to put the Employee class definition in the models directory. Rails uses a generator to create the model for the employees table, which you use by navigating to your application directory then typing ruby script/generate model employee in the **Ruby Console**. The result is shown in Fig. 24.17.

The last line the console returns is create db/migrate/001_create_employees.rb. We have not yet created a table employees, so Ruby automatically generates a script that will create this table when the application launches. We can modify this script to perform additional initial changes to the employees table. Figure 24.19 shows a modification of 001_create_employees.rb (located in your application's db/migrate directory) that creates the table and adds three records to it.

ActiveRecord has a special feature called Migration, which allows you to preform database operations within Rails. Each object that inherits from ActiveRecord::



```
C:\InstantRails\rails_apps\Employees>ruby script/generate model employee
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
   identical  app/models/employee.rb
   identical  test/unit/employee_test.rb
   identical  test/fixtures/employees.yml
      exists  db/migrate
      create  db/migrate/001_create_employees.rb

C:\InstantRails\rails_apps\Employees>_
```

Fig. 24.17 | Creating a model in the Ruby Console.

```
 1  # Fig. 24.18: db/migrate/001_create_employees.rb
 2  # Database migration script modified to add data to the table
 3  class CreateEmployees < ActiveRecord::Migration
 4     # create the table with three columns and insert some rows.
 5     def self.up
 6        create_table  :employees do |t|
 7           t.column :first_name, :string
 8           t.column :last_name, :string
 9           t.column :job_title, :string
10        end # do block
11
12        Employee.create :first_name => "Sue", :last_name => "Green",
13           :job_title => "Programmer"
14        Employee.create :first_name => "Meg", :last_name => "Gold",
15           :job_title => "Programmer"
16        Employee.create :first_name => "John", :last_name => "Gray",
17           :job_title => "Programmer"
18     end # method self.up
19
20     # reverse the migration, delete the table that was created
21     def self.down
22        drop_table :employees
23     end # method self.down
24  end # class CreateEmployees
```

Fig. 24.18 | Database migration script modified to add data to the table.

Migration must implement two methods—self.up (lines 5–18), which preforms a set of database operations, and self.down (lines 21–23), which reverses the database operations performed in self.up. In this case self.up creates the table with three columns and adds data to it, and self.down deletes the table. Line 6 calls the create_table function passing as a parameter a code block, inside the do, containing the table's column names and types. Lines 12–17 use ActiveRecord's built in create method to add data to the Employees table. ActiveRecord has built-in functionality for many create, retrieve, update, and destroy methods—known in Rails as CRUD. These methods represent the trivial operations that you would want to do with a database.

We can execute the migration using Ruby's rake command. To do so open up the **Ruby Console**, navigate to your application's directory and type rake db:migrate. This command will call the self.up method of all the migrations located in your db/migrate directory. If you ever want to roll back the migrations you can type in rake db:migrate VERSION=0, which calls each migration's self.down method. Specifying a version number other than 0 will call the self.down method of all the migrations whose number is greater then the version number.

Common Programming Error 24.1

If the code that comes after the creation of the table in the self.up *is erroneous, the migration will fail, and will not be able to execute again because the table will already exist. Also, Rails will not have marked the migration as successfully completed, so the version will still be 0 and the migration cannot be rolled back. One way to prevent this problem is to force the table to be dropped every time before creating it. Another solution is splitting up the migration into smaller discrete migrations, one to create the table and another to insert data in the table.*

Because our model will never be modified by the application, we do not need to add any functionality to it. Figure 24.19, which represents the `employee.rb` file located in the `app/Models` directory, contains all the code that is needed to integrate the `employees` database table into the application.

```
# Fig. 24.19: employee.rb
# Generated code for an Employee Model.
class Employee < ActiveRecord::Base
end # class Employee
```

Fig. 24.19 | Generated code for an `Employee` model.

Employee Controller

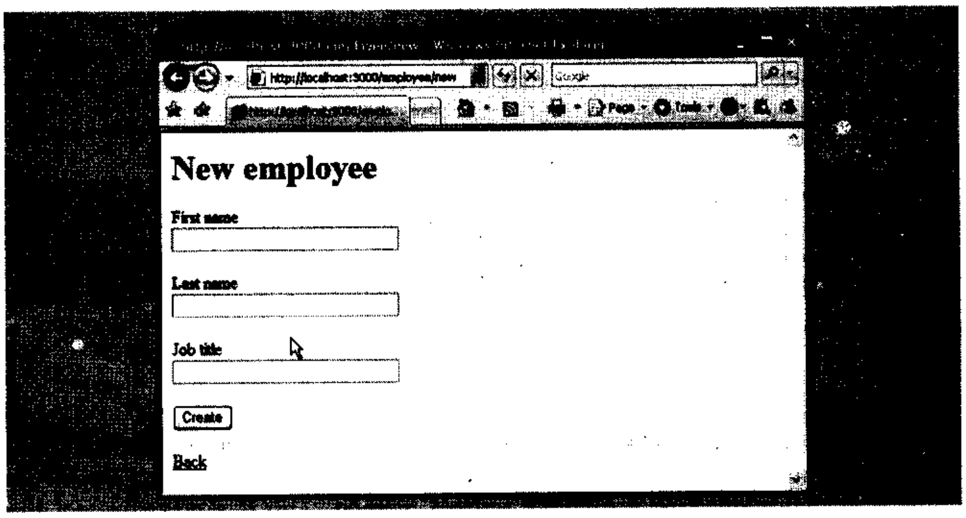
Next, create the controller with the `ruby script/generate controller employees` command as shown in Section 24.4. Figure 24.20 shows the example controller for the Employee application. Line 4 calls the **scaffold method**. This is a powerful tool that automatically creates CRUD functionality. It creates methods such as `new`, `edit` and `list` so you don't have to create them yourself. It also defines default views for these methods that are rendered when each method is called. You can override the default functionality by defining your own methods. If you override all the CRUD methods you can delete the `scaffold` method. When you override a method, you must also create the corresponding view. Since we will not modify the new method created by the scaffold you can see the new method's view with the URL `http://localhost:3000/employee/new` (Figure 24.21). Line 7–9 override the `list` method. Line 8 queries the database and returns a list of all of the `Employee` objects, which gets stored in an `@employees` instance array. This data will be passed to the view.

The `list` View

The `list` template is rendered by the `list` method from the `EmployeeController`. Code for the `list` template is shown in Fig. 24.22. This file should be placed in your application's `app/views/employee` directory. While most of it is just standard XHTML, lines 14–17 contain Ruby code that iterates through all the employees in the `@employees` array instance variable, and outputs each employee's first and last name (line 16). A `for` statement like this in a `list` view is common in database-driven web applications.

```
 1  # Fig. 24.20: app/controllers/employees_controller.rb
 2  # Provides all of the functionality for the application.
 3  class EmployeesController < ApplicationController
 4     scaffold :employee # create scaffold code for controller
 5
 6     # override scaffold list method
 7     def list
 8        @employees = Employee.find( :all ) # return an array of Employees
 9     end # method list
10  end # class EmployeeController
```

Fig. 24.20 | `Employee` controller provides all of the functionality for the application.



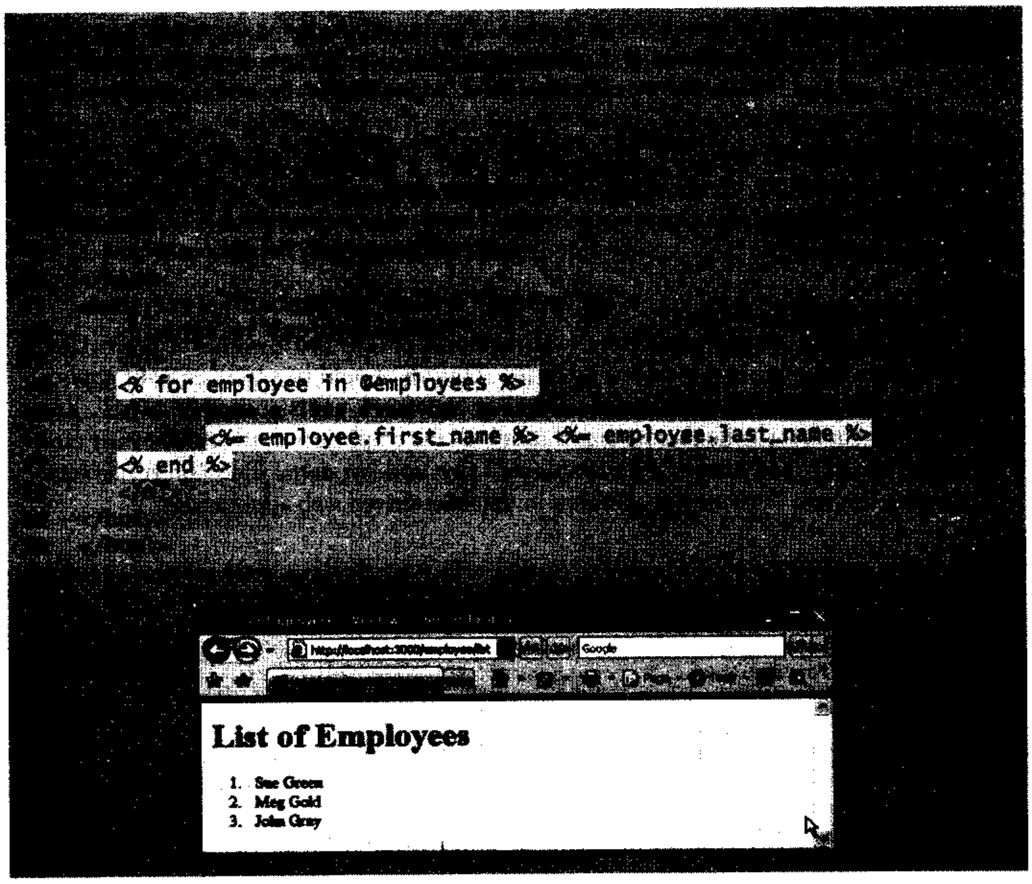
New employee

First name

Last name

Job title

Fig. 24.21 | View of the new action when generated by the `scaffold`.



```
<% for employee in @employees %>
      <%= employee.first_name %> <%= employee.last_name %>
<% end %>
```

List of Employees

1. Sue Green
2. Meg Gold
3. John Gray

Fig. 24.22 | A view that displays a list of `employees`.

24.6 Case Study: Message Forum

Our next example uses Ruby on Rails to create a **message forum** website. Message forums enable users to discuss various topics. Common features of message forums include discussion groups, questions and answers and general comments. To see some popular message forums, visit `messages.yahoo.com`, `web.eesite.com/forums` and `groups.google.com`. In this example, users can post messages to several different forums, and administrators of the message forum site can create and delete forums.

Design

For our message forum application, we need a table containing all of the messages. This table will be called `messages` and will contain attributes such as `id`, `title`, `author`, `e-mail`, `created_on` (the date the message was created) and `forum_id` (the `id` of the forum to which the message belongs). In addition, we need a table of all the available forums. This table, called `forums`, will contain attributes such as `id`, `name`, `administrator` and `created_on` (the date the forum was created).

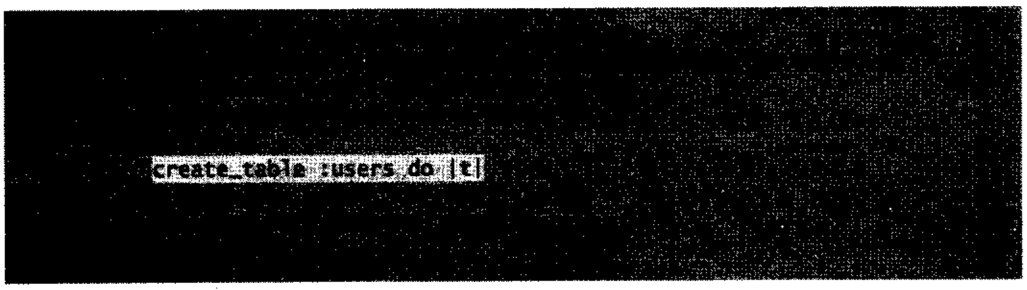
In our message forum application, we want to have the functionality to create and delete forums, but we don't want everyone who uses our application to be able to do this. Therefore, we will also have a `users` table, which contains the username/password combinations of all the application's administrators.

Before we implement this design we must create the empty application called `messageboard` and the database for this application. Type in `rails Messageboard` and then `mysqladmin -u root create messageboard_development` in the **Ruby Console**.

24.6.1 Logging In and Logging Out

Use the `model` generator to generate the `User` model by typing `ruby script/generate model User` into the **Ruby Console** (from the `Messageboard` directory). Next, create the table that will be associated with the model. To do that, modify the migration created by the model generator to set up the users table and add some data to it. Figure 24.23 is the `001_create_users.rb` migration (from the `db/migrate` directory) which sets up the user table.

The `create_table` function call (lines 6–9) specifies the table's columns. By default a primary key `id` column is created, so it is not included here. Lines 7–8 create the `name` and `password` columns with appropriate types. Note that the `name` has a `limit` of 11 characters. Line 11 adds data to the table. To execute this migration type `rake db:migrate` in the **Ruby Console**.



```
create_table :users do |t|
```

Fig. 24.23 | Database migration script modified to add data to the table. (Part 1 of 2.)

```

      User.create :name => "user1", :password => "54321"
   end # method self.up

   # remove users table
   def self.down
      drop_table :users
   end # method self.down
end # class CreateUsers
```

Fig. 24.23 | Database migration script modified to add data to the table. (Part 2 of 2.)

Common Programming Error 24.2

Creating a column without explicitly specifying a limit on length will cause Rails to truncate the data entered into the database with database-defined limits.

Since the users table never changes, nothing needs to be specified in the User model, but one still needs to exist. This will allow the controller to access a User as an ActiveRecord. Figure 24.24 shows the empty model for the users table.

Next, we need to provide user creation, field validation and user logout functionality through the use of a controller (Fig. 24.25). Create this controller by typing ruby script/generate controller user. When a user logs in, we will keep track of that User object in a session variable—a variable that maintains information across multiple pages of a web application. The purpose of the admin method (lines 5–7) is to pass a blank user object into the admin page, which will get filled with information, then render the admin.rhtml template. The validate method (lines 10–21) checks the user model to determine

```
# Fig. 24.24: app/models/user.rb
# Generated code for the User model.
class User < ActiveRecord::Base
end # method User
```

Fig. 24.24 | Generated code for the User model.

```
 1  # Fig. 24.25: app/controllers/users_controller.rb
 2  # UsersController provides validation functionality for the table.
 3  class UsersController < ApplicationController
 4     # create a new User object
 5     def admin
 6        @user = User.new # create a new User object
 7     end # method admin
 8
 9     # validate that user exists
10     def validate
11        # find a user with the correct name and password
12        @user = User.find_by_name_and_password( params[ :user ][ :name ],
13           params[ :user ][ :password ] )
14
```

Fig. 24.25 | UsersController provides validation functionality for the table. (Part 1 of 2.)

```
  if ( @user == nil ) # if the user dosn't exist
    redirect_to :action => "admin" # redirect to admin action
  else # user does exist
    session[ :user ] = @user # store the user in a session variable
    redirect_to :controller => "forums", :action => "index"
  end # if
end # method validate
[illegible]
   reset_session # delete all session variables
   redirect_to :controller => "forums", :action => "index"
[illegible]
```

Fig. 24.25 | `UsersController` provides validation functionality for the table. (Part 2 of 2.)

whether the username exists, then redirects the application to the next action based on the result of that check.

Rails allows us to generate methods dynamically to serve a specific purpose. Lines 12–13 call the `find_by_name_and_password` method, which searches the model with the `name` and `password`, passed as a parameter.

The `validate` method assigns to an instance variable named `@user` (line 12) the value of the `User` that was returned by the `find_by_name_and_password` method. If no such `User` exists, the client is redirected to the `admin` page and asked to log in again (line 16). If the `User` does exist, a session variable is created (line 18), and line 19 redirects the client to the `index` of the `forums` controller, which we create in Section 24.6.4. The `logout` method (lines 24–27) uses the `reset_session` method to delete all the user's session variables, forcing the user to sign in again to use administrative options.

Performance Tip 24.1

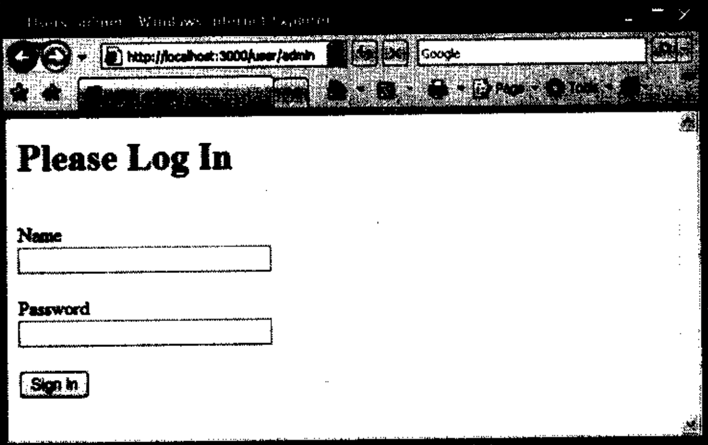
Storing full objects in the session is inefficient. The user object is one of the rare exceptions, because it doesn't change very often and is frequently needed in web applications that manage the state information for unique clients.

Method `admin`'s view is a simple login form. The template is shown in Fig. 24.26. It asks the user for a name and password using the **`text_field`** (line 6) and **`password_field`** (line 9) helpers, then sends the information to the `validate` method when the user clicks **Sign In**.

```
[illegible]
<% form_tag :action => 'validate' do %> [illegible]
  <%= text_field 'user', 'name' %> [illegible]
```

Fig. 24.26 | Login form used to send data to the `user` controller. (Part 1 of 2.)

```
    <%= password_field 'user', 'password' %>
    <%= submit_tag "Sign In" %>
  <% end %>
```



Please Log In

Name

Password

Sign In

Fig. 24.26 | Login form used to send data to the `user` controller. (Part 2 of 2.)

Rails **helpers** are methods that generate XHTML content for the view. The `password_field` helper method generates a text field that masks the text inside it. Both `text_field` and `password_field` specify a model and the column. This information is used to determine the validation properties for each column when validating the text typed into these fields. When the user clicks the submit button defined in line 10, the **`form_tag`** method (line 4) automatically generates a `Hash`, where the keys are the names of the input fields and the values are what the user entered, and sends it to the `validate` action. The link to the `validate` action is specified by the `action` option. To display this action, run the Mongrel server and navigate your browser to `http://localhost:3000/user/admin`.

We define the `user` controller's template in Fig. 24.27. Because the `user` controller has only a single view to render, we could have simply include this XHTML in the view. The benefit of a template is that it allows us to easily add more views in the future that are all based on the same template and adhere to Ruby on Rails' DRY (Don't Repeat Yourself) philosophy. Line 9 displays the current action in the title bar. Line 12 is the placeholder for the content of the action's view.

[illegible]

Fig. 24.27 | Display the name of the current action in the title bar. (Part 1 of 2.)

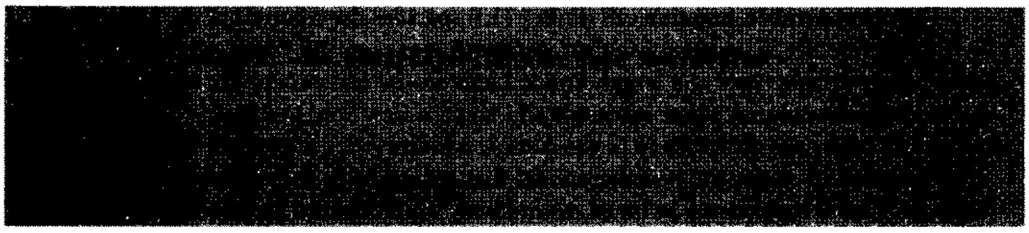


Fig. 24.27 | Display the name of the current action in the title bar. (Part 2 of 2.)

24.6.2 Embellishing the Models

Several methods must be added to the model so that it can be modified from the application. These methods are all defined by `ActiveRecord`.

Message Model

First, we must create an empty `Message` model by typing `ruby script/generate model Message` in the **Ruby Console**. Before we make any change to the model we must create the `messages` table. Figure 24.28 is the migration that creates the messages table and adds data to it. To run this migration, navigate to the `messageboard` directory and type `rake db:migrate`.

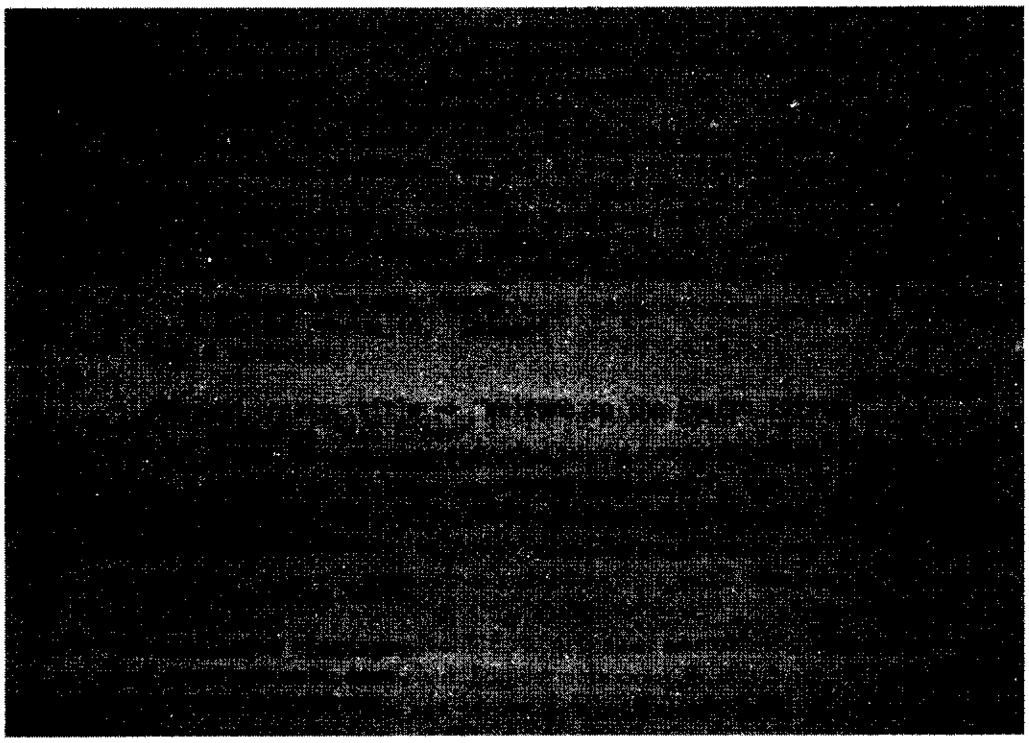
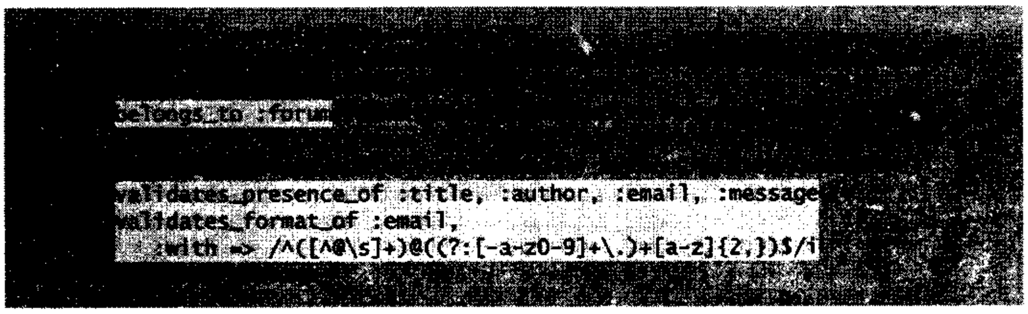


Fig. 24.28 | Database migration script modified to add data to the table.

The `create_table` function call (lines 6–13) specifies the columns of the table. Lines 7–12 create the `title`, `author`, `created_on`, `email`, `message` and `forum_id` columns with appropriate variable types and length limits. Lines 15–20 add data to the table. Rails fills the `created_on` column value automatically when a new row is created. To apply the migration, type `rake db:migrate` in the console window.

Figure 24.29 shows the `Message` model that encapsulates the `messages` table in the database. Line 4 invokes the `belongs_to` method, which defines an association with the `forums` table that can be used to access elements of the `forums` table. This method will allow the `Message` to access the `forum` to which the given `Message` belongs simply by calling a method named `forum` on the `Message`. This is known as an **association method**.

Lines 7–9 are examples of **validators** that can be applied to an object that inherits from `ActiveRecord`. These validations occur when the `save` method is called on a `message` object in an attempt to store it in the database. If the validations are successful, then the object is saved to the database and the method returns `true`. If the validations fail, an `Errors` object associated with the `Message` object is updated and the method returns `false`. The method **`validates_presence_of`** ensures that all of the fields specified by its parameters are not empty. The method **`validates_format_of`** matches all of the fields specified by its parameters with a regular expression. The regular expression in line 9 represents a valid e-mail address. This regular expression can be found in the Rails framework documentation at `api.rubyonrails.org`.



```
belongs_to :forum

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

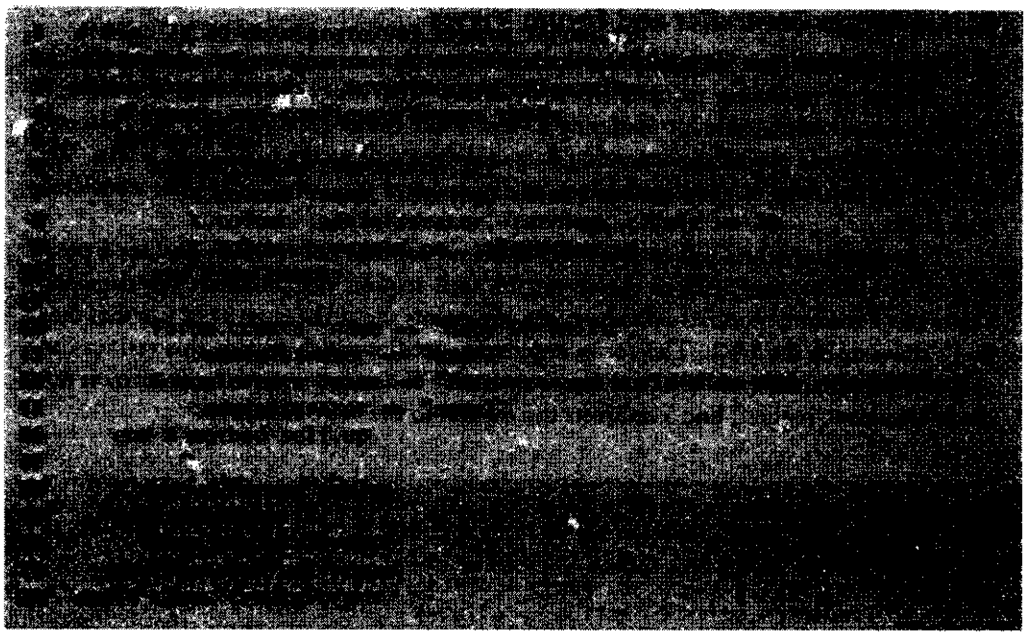
Fig. 24.29 | `Message` model containing validation and initialization functionality.

Forum Model

Next, create an empty `forum` model by typing in `ruby script/generate model forum`. Then create the `forums` table in a similar fashion to `messages` and `users`. Figure 24.30 is the `Migration` that sets up the `messages` table.

The `create_table` function call (lines 6–10) specifies the columns of the table. Lines 7–9 create the `name`, `administrator` and `created_on` columns with appropriate variable types and length limits. Lines 12–16 add data to the table. To apply the migration, type in `rake db:migrate`.

The model for the `forums` table (Fig. 24.31) looks similar to the model for the `messages` table. We can create an association method that allows a `Forum` to access every `Message` that is associated with it. Line 4 shows the **`has_many`** method, which will create a method called `messages` for every `Forum` object. The `messages` method will return an array of all the `messages` in the `Forum`.



[illegible]

Fig. 24.30 | Database migration script modified to add data to the table.

```
has_many :messages, :dependent => :destroy
```

Fig. 24.31 | Forum model that includes validation and initialization functionality.

When a `forum` is deleted, all of that `forum`'s `messages` should also be deleted. Line 4 sets the `dependent` parameter of the `has_many` method to `:destroy` to ensure that when a forum is destroyed all the messages that are associated with it are destroyed as well.

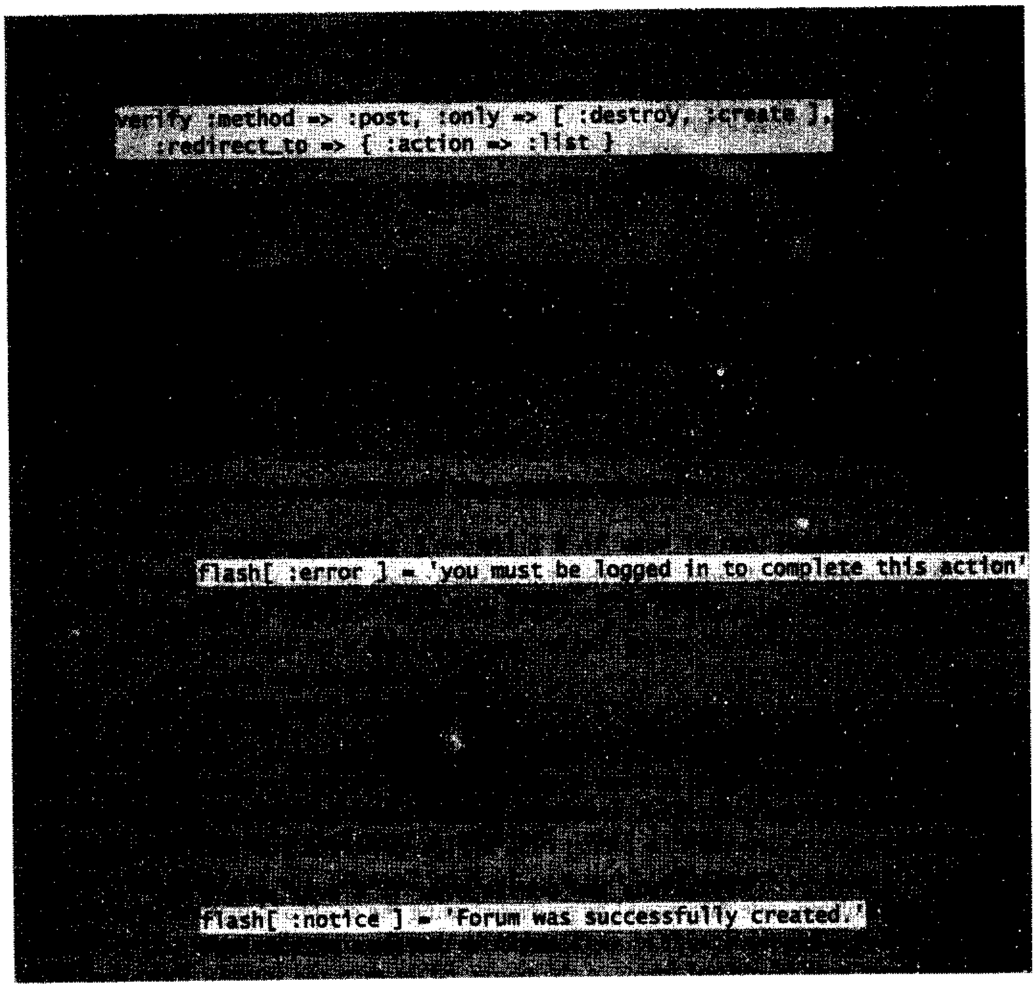
24.6.3 Generating Scaffold Code

Now that the user can log in and out of our application, we need to create the `messages` and `forums` views and controllers. Since much of this code is standard CRUD, we can use the Rails `scaffold` generator by typing `ruby script/generate scaffold message` and `ruby script/generate scaffold forum` in the **Ruby Console**. The `scaffold` generator creates the scaffold code that would be generated using the `scaffold` method in the controller. When using the `scaffold` method, notice that the controller name and the view directory are both pluralized forms of the name, rather than the singular form that the model generator and controller generator would create.

24.6.4 Forum Controller and Forum Views

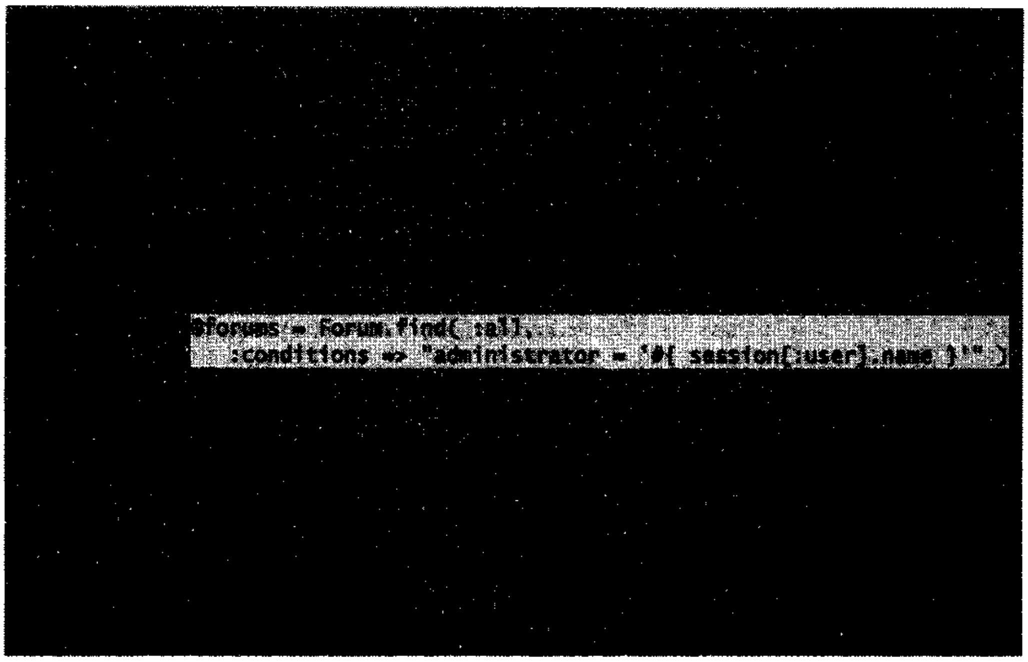
The `ForumsController` (Fig. 24.32), which was initially generated as part of the scaffolding, handles all incoming requests from the client. The `index`, `list`, `new`, and `delete` methods are all responsible for rendering a view, while the `create` and `destroy` methods are responsible for processing incoming data and then redirecting to another action. We will not use the `edit` or `show` methods so you may delete the `.rhtml` view files associated with them.

The `verify` method call in lines 4–5 is edited scaffold code, which ensures a `post` request is used to send data to the server for each request that modifies the database. Whenever a method modifies a database, the arguments should be from a `post` so that they don't show up in the URL. The `:only` argument specifies which actions this verification should be applied (`create` and `destroy` in this case). If a call to `create` or `destroy` is not made via a `post` request, line 5 redirects the request to the `list` action.



```
verify :method => :post, :only => [ :destroy, :create ],
   :redirect_to => { :action => :list }
[illegible]
flash[ :error ] = 'you must be logged in to complete this action'
[illegible]
flash[ :notice ] = 'Forum was successfully created.'
[illegible]
```

Fig. 24.32 | `ForumsController` implements CRUD functionality. (Part 1 of 2.)



```
@forums = Forum.find( :all,
  :conditions => "administrator = '#{ session[:user].name }'" )
```

Fig. 24.32 | `ForumsController` implements CRUD functionality. (Part 2 of 2.)

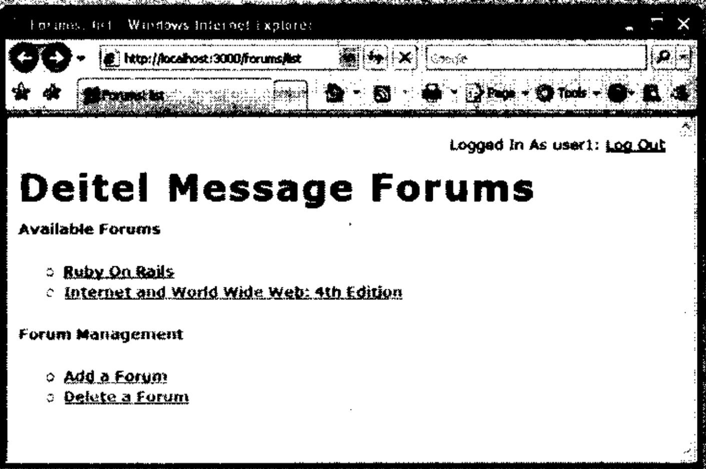
Method `index` (line 8–11) redirects the client to the `list` method (lines 14–16), which obtains a list of forums from the database to be displayed on the page. The `new` method (lines 19–26) checks whether the user has privileges to create a new forum. If not, lines 21–22 display an error and redirect the user to the `index` action. The hash called `flash` in line 21 is used to display `messages` in the view. `Flash` is a special type of session storage that is always automatically cleared after every request to the controller. If the user has privileges, line 25 creates a new instance of the forum object which is initialized with data from the user input.

The `create` method (lines 29–39) is similar to the scaffold code, but differs in that the `administrator` attribute of the `forum` being saved must be the `name` of the user who is logged in. Line 33 attempts to save the forum, and either renders a template if the method returns `false` (line 37), or redirects to the `list` method and updates the `flash` object if the method returns `true` (lines 34–35). The `delete` method (lines 42–50) sets up the deletion operation by finding all the forums created by the user currently logged in. Once the user picks a forum to delete in the view, the `destroy` method (lines 53–58) destroys the forum specified by the user (line 55) and re-displays the list (line 56).

List View

Figure 24.33 is the template that is rendered by the `list` method from the `Forum` controller. [*Note:* We replaced the auto-generated `list.rhtml` from the scaffolding]. This is also the template rendered by the `index` method. Line 10 uses the `link_to` method to create a link with the name of the forum and the `list` action of its messages, which we build in Section 24.6.5.. Lines 12–13 contain a conditional statement which make the forum italicized for five minutes after it has been created by using the `minutes.ago` method of the

```
[illegible]
```



Forums: list - Windows Internet Explorer

http://localhost:3000/forums/list

Logged In As user1: Log Out

Deitel Message Forums

Available Forums

- Ruby On Rails
- Internet and World Wide Web: 4th Edition

Forum Management

- Add a Forum
- Delete a Forum

Fig. 24.33 | Template for the list action that displays a list of forums.

`Fixnum` class. The `if` statement in lines 17–24 displays the XHTML in lines 19–23 only if there is a user logged in.

New View

Figure 24.34 shows the template for the `Forum` controller's new method. This is code generated by the `scaffold`. Lines 4–7 create a form that is rendered to the page, and indicate

```
[illegible]
```

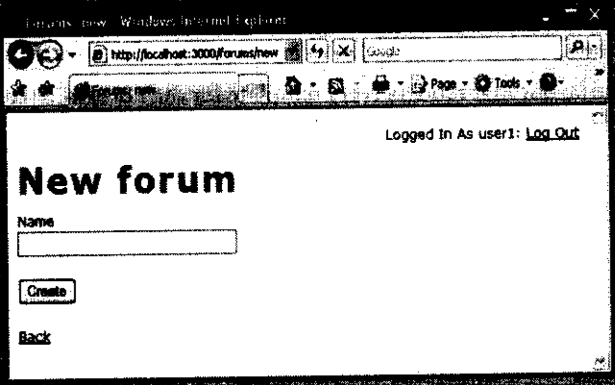
Fig. 24.34 | Template for a new action that adds a `forum` to the `forums` table.

that the action `create` will be called when the **Create** button is pressed. This template renders a **partial**—a block of HTML and embedded Ruby code stored in another file and inserted directly into the document. A partial allows the same block of code to be used across multiple documents. In this example, line 5 renders the partial named `form`, which inserts the file `_form.rhtml` at that line of code. A partial filename always begins with an underscore.

Line 6 uses the `submit_tag` method to create a submit button that when clicked will create a `Hash` with the form's fields as keys and the user's input as values. Line 9 uses the `link_to` function to allow the user to go back to the forums list by redirecting the client to the `list` action of the `forum` controller.

The partial in Fig. 24.35 renders the input text field for the form. The `administrator` and `created_on` fields will be generated on the server, so they've been deleted from the scaffold's code. The `created_on` field will be automatically set to the time when the forum is created. The administrator field will be set by the controller.

```
<%= error_messages_for 'forum' %>
[illegible]
```


Logged In As user1: Log Out

New forum

Name

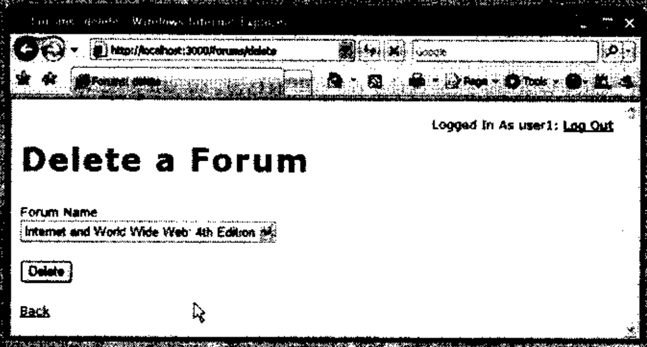
Forums: new - Windows Internet Explorer
http://localhost:3000/forums/new
Google
Page
Tools
Create
Back

Fig. 24.35 | Partial that contains a form used to add a new `forum`.

Delete View

The `delete` view (Fig. 24.36) is not generated by the scaffold, so we create it ourselves. It is similar to `create` in that it renders a form, but uses the `collection_select` method to display for that administrator the list of available forums to delete. The **`collection_select`** takes five parameters—the type of object to be selected, the field which the options are to be grouped by, the collection from that to obtain the list of objects, the field that will be sent once an option is selected and the field that is to be displayed on the screen for each option.

```
<!-- Fig. 24.36: app/views/forums/delete.rhtml -->
<!-- Template for delete action used to delete a forum. -->
[illegible]
[illegible]
[illegible]
[illegible]
   <%= collection_select "forum", "id", @forums, "id", "name" %>
[illegible]
[illegible] close end form tag [illegible]
[illegible]
```


Logged In As user1: Log Out

Delete a Forum

Forum Name

Internet and World Wide Web 4th Edition

Delete

Back

Fig. 24.36 | Template for `Delete` action used to delete a `forum`.

Forum Layout

Figure 24.37 is the layout that renders every template for the `ForumsController`. It has all the necessary XHTML, and contains the login/logout text (lines 13–25). Line 29 automatically renders the template of any action that uses the template.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 24.37: app/views/layouts/forums.rhtml -->
<!-- Layout that displays the logged-in user for every Forums action. -->
```

Fig. 24.37 | Layout that displays the logged-in user for every `Forums` action. (Part 1 of 2.)

```
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <title>Forums: <%= controller.action_name %></title>
   <%= stylesheet_link_tag 'scaffold' %> <!-- link to a stylesheet -->
[illegible]
[illegible]
   <div style = "text-align: right">
      <% if ( session[ :user ] ) then %> <!-- if user is logged on -->
         <!-- code to display if user is logged on -->
         <%= "Logged In As #{ session[ :user ].name }: " %>
         <%= link_to 'Log Out',
            :controller => 'users', :action => 'logout' %>
      <% else %> <!-- user is not logged on -->
         <!-- code to display if user is not logged on -->
         <%= "Not Currently Logged In:" %>
         <%= link_to 'Log In',
            :controller => 'users', :action => 'admin' %>
      <% end %> <!-- end if -->
   </div>
   <p style="color: green"><%= flash[ :notice ] %></p>
   <p style="color: red"><%= flash[ :error ] %></p>

   <%= yield %> <!-- displays template -->
[illegible]
[illegible]
```

Fig. 24.37 | Layout that displays the logged-in user for every Forums action. (Part 2 of 2.)

If the user is logged in (line 14), lines 16–18 display the username on the page and use the link_to method to enable the user to log out by redirecting to the logout action. Otherwise, lines 21–23 allow the user to log in, using the link_to helper method to redirect the user to the admin action. Lines 26–27 display any error messages or success messages that result from user interactions.

24.6.5 Message Controller and Message Views

The MessagesController (Fig. 24.38) is similar to the ForumsController, except that its list method (lines 8–22) doesn't list all of the messages—it lists only the ones with the specified forum_id that is passed in as a URL parameter from the Forum list view. Line 10 updates the session variable to match the URL parameter. If no parameter value is specified, the forum_id session variable is used. If neither of these exists, line 14 displays an error and line 15 redirects the client to the list action of the forum controller. The find method which is called on Message (line 18–19) specifies that the messages should be ordered by their created_on dates in descending order. Line 20 also calls the find method to obtain the forum object, which we will need to add messages to the forum.

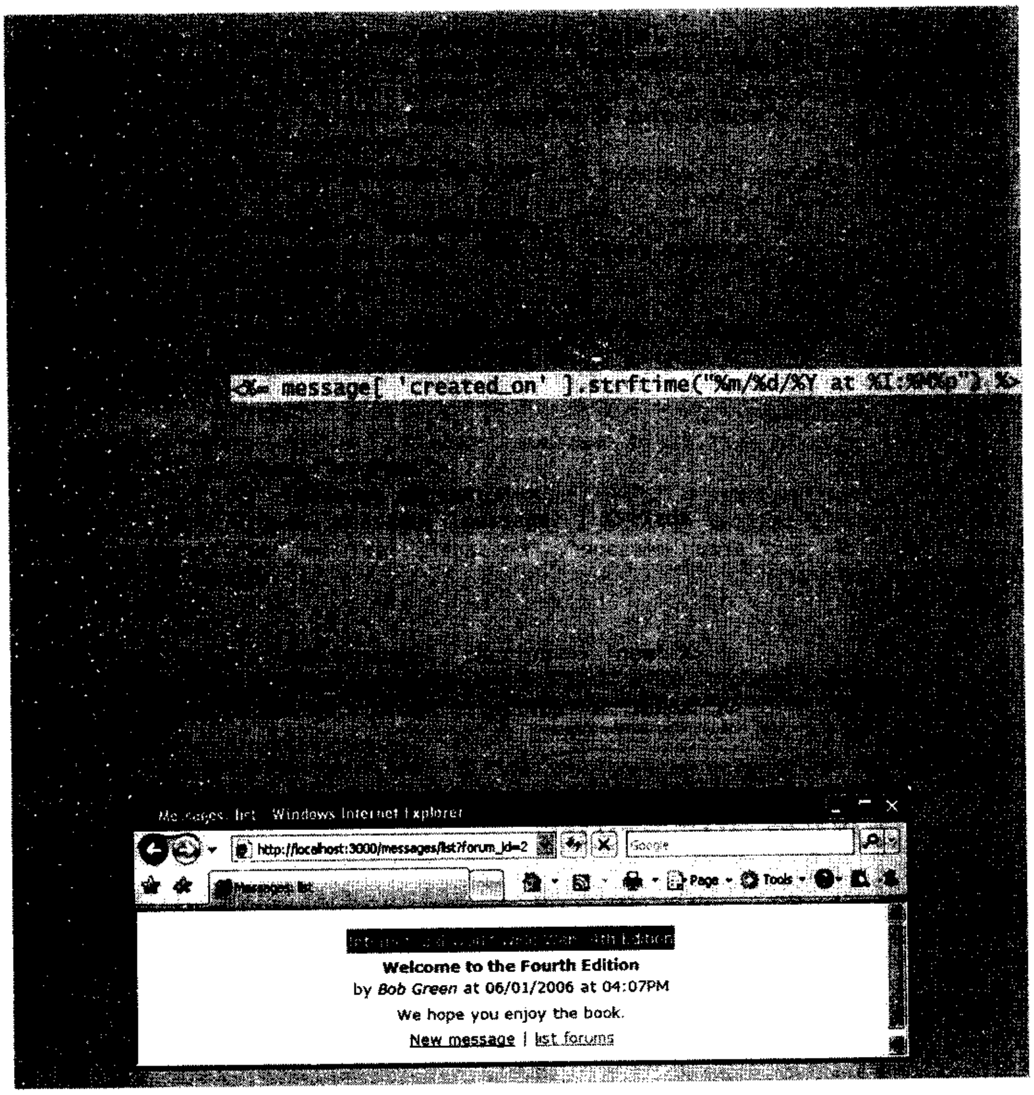
The create method (lines 30–40) replaces the method generated by the scaffold. Line 31 obtains the id of the forum to which the message should be added and line 32 obtains the new message entered by the user. If the message is added to the database successfully, lines 35–36 set the appropriate message to be displayed in the view using the flash object and redirect the client to the list action. Otherwise, line 38 redirects the client to the new action, prompting the user to enter the message again.

[illegible]

Fig. 24.38 | `MessagesController` that implements CRUD functionality.

List View

The `list` view (Fig. 24.39) for the `Message` controller is similar to the `list` view for the `Forum` controller, except that more information is displayed in the messages `list` view. It uses CSS to format the output. In this view, every `message` object acts like a `Hash`—passing a column name as a key returns the corresponding value in the `message` object. To obtain an a column's value, include the attribute method's name in square brackets after the name of the object. For each `message` in the forum, line 12 displays the title, line 13 displays the author and line 19 displays the message's text. At line 14, the Ruby `Time` object that is returned by the `message['created on']` is formatted using the Ruby `Time` class formatting



```
<%= message[ 'created_on' ].strftime("%m/%d/%Y at %I:%M%p") %>
```

Messages: list - Windows Internet Explorer
http://localhost:3000/messages/list?forum_id=2
Google
Messages: list
Page
Tools

Welcome to the Fourth Edition
by *Bob Green* at 06/01/2006 at 04:07PM
We hope you enjoy the book.
New message | list forums

Fig. 24.39 | Template for the `list` action that displays a list of messages.

options. Lines 24–26 use the `link_to` method to allow the user to create a message in the current forum or to go back to the list of the forums.

New View

The `new` template for the `Message` controller is omitted here because it is scaffold code that is nearly identical to the `new` template for the `Forum` controller. The partial shown in Fig. 24.40 for the `messages` form is also similar. Lines 8, 12 and 16 use the `text_field` helper method to create fields for specifying the `title`, `author` and `email`. Line 20 uses the `text_area` helper method to create an input area of a certain size, to be used to input the message. These fields are validated when the `Message` model's `save` method is called. If the model does not deem the data valid, line 3 displays the error messages.

```
<!-- Fig. 24.40: app/views/messages/_form.rhtml -->
<!-- A form that allows the user to enter a new message. -->
<%= error_messages_for 'message' %>

<table>
   <tr class = "alignRight">
      <td>Title</td>
      <td><%= text_field 'message', 'title' %></td>
   </tr>
   <tr class = "alignRight">
      <td>Author</td>
      <td><%= text_field 'message', 'author' %></td>
   </tr>
   <tr class = "alignRight">
      <td>Email</td>
      <td><%= text_field 'message', 'email' %></td>
   </tr>
   <tr><td colspan = "2">Message</td></tr>
   <tr><td colspan = "2">
      <%= text_area 'message', 'message', :cols => "30", :rows => "4"%>
   </td></tr>
</table>
```

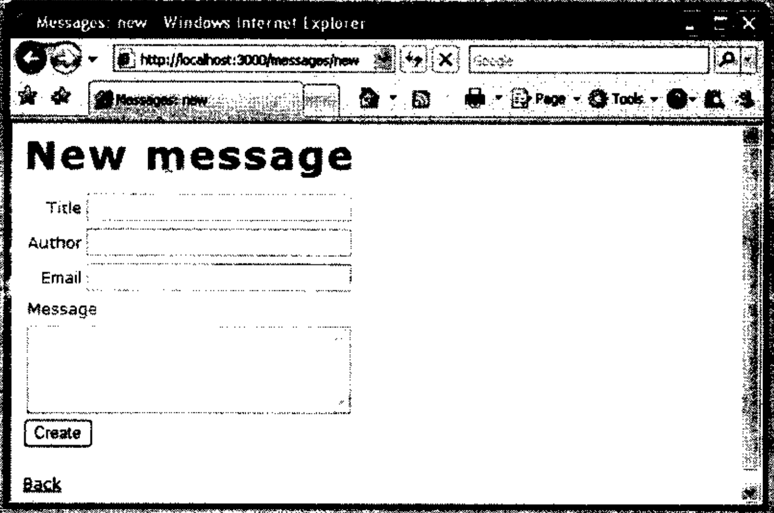

Messages: new - Windows Internet Explorer
http://localhost:3000/messages/new
New message
Title
Author
Email
Message
Create
Back

Fig. 24.40 | Form that allows the user to enter a new message.

Message Layout

Figure 24.41 shows the layout used to render all `Message` templates. Line 10 invokes the `scaffold.css` style sheet, which we changed slightly to improve our page's presentation. To make the style sheet available for import it must be placed in the `public/stylesheets` directory of the application. When a forum has been modified, line 14 displays the appropriate message.

```
[illegible]
<%= stylesheet_link_tag 'scaffold' %>
[illegible]
```

Fig. 24.41 | Message layout that links a style sheet and displays a message.

We have now implemented the basic functionality of the forum application. To test this application execute it on Mongrel and browse to `http://localhost:3000/forums`. To test the administrative privileges of the forum go to `http://localhost:3000/user/admin` and login with the username `user1` and password `54321`. In the next section we'll add Ajax capabilities to make our forum more responsive.

24.6.6 Ajax-Enabled Rails Applications

Adding Ajax functionality to Rails applications is straightforward. Rails includes a JavaScript library called Prototype that contains easy-to-use cross-browser Ajax functions. Figure 24.42 is the modified layout for the forum file, which now links the prototype library to the application. For the application to have the correct look, make sure you insert the modified style sheet, which can be found in our examples folder, into the `public/stylesheets` directory of the application.

```
[illegible]
<%= javascript_include_tag :defaults %>
[illegible]
```

Fig. 24.42 | Forums layout that uses the default JavaScript libraries. (Part 1 of 2.)

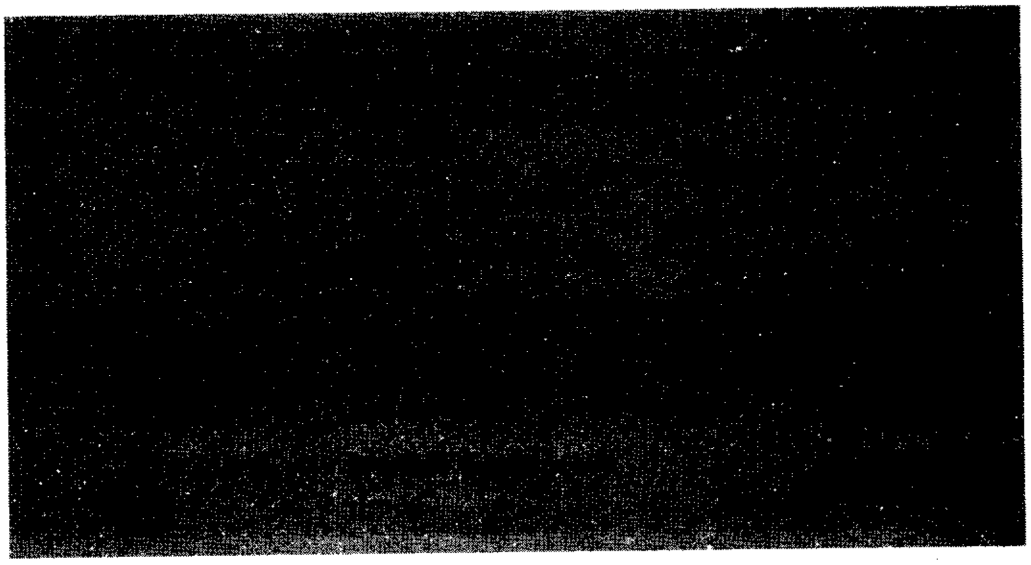
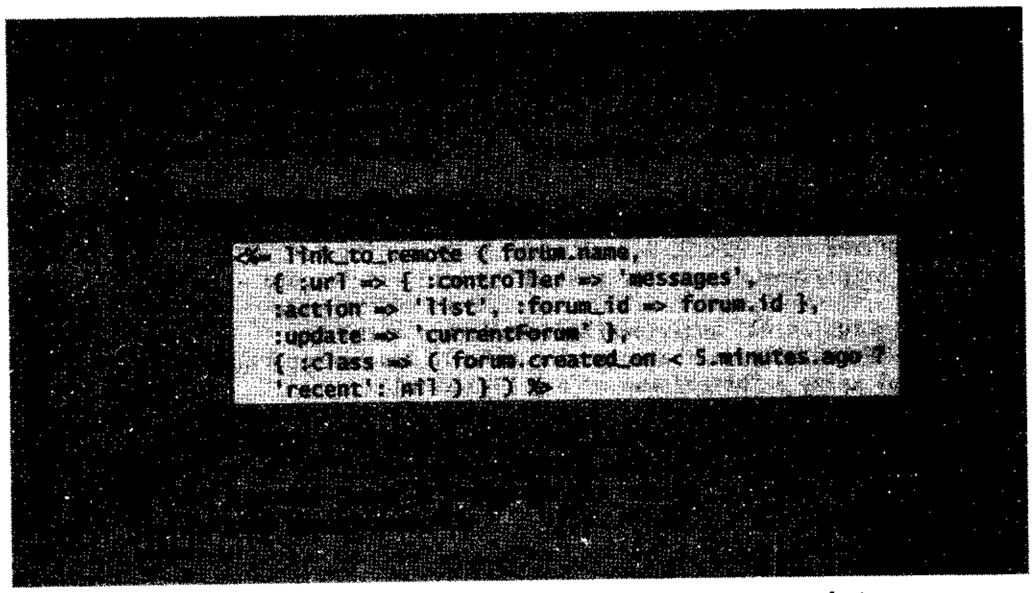


Fig. 24.42 | Forums layout that uses the default JavaScript libraries. (Part 2 of 2.)

Line 11 links in the JavaScript library using the **javascript_include_tag** helper method. The defaults parameter tells javascript_include_tag to link all the **defaults** JavaScript Rails libraries including Prototype and Script.aculo.us. The rest of the layout file is the same as in the non-Ajax version.

Figure 24.43 changes the Forum object's list view to perform Ajax requests rather than load a new page. Now, whenever the user clicks a forum's name, the page loads the forum's messages to the right of the forums list with a partial page update.

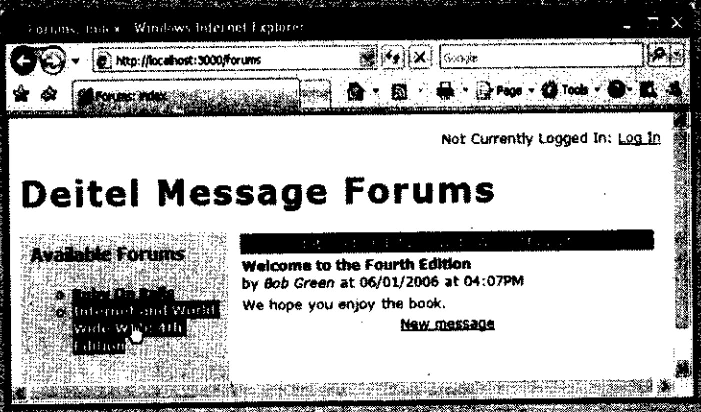


```
<%= link_to_remote ( forum.name,
  { :url => { :controller => 'messages',
  :action => 'list', :forum_id => forum.id },
  :update => 'currentForum' },
  { :class => ( forum.created_on < 5.minutes.ago ?
  'recent': nil ) } ) %>
```

Fig. 24.43 | Displaying a list of messages without reloading the page. (Part 1 of 2.)

```
[illegible]

<div id = 'currentForum' class = "ajaxComponent">
</div>
```



Not Currently Logged In: Log In

Deitel Message Forums

Available Forums

- Ruby On Rails
- Internet and World Wide Web 4th Edition

Welcome to the Fourth Edition
by Bob Green at 06/01/2006 at 04:07PM
We hope you enjoy the book.
New message

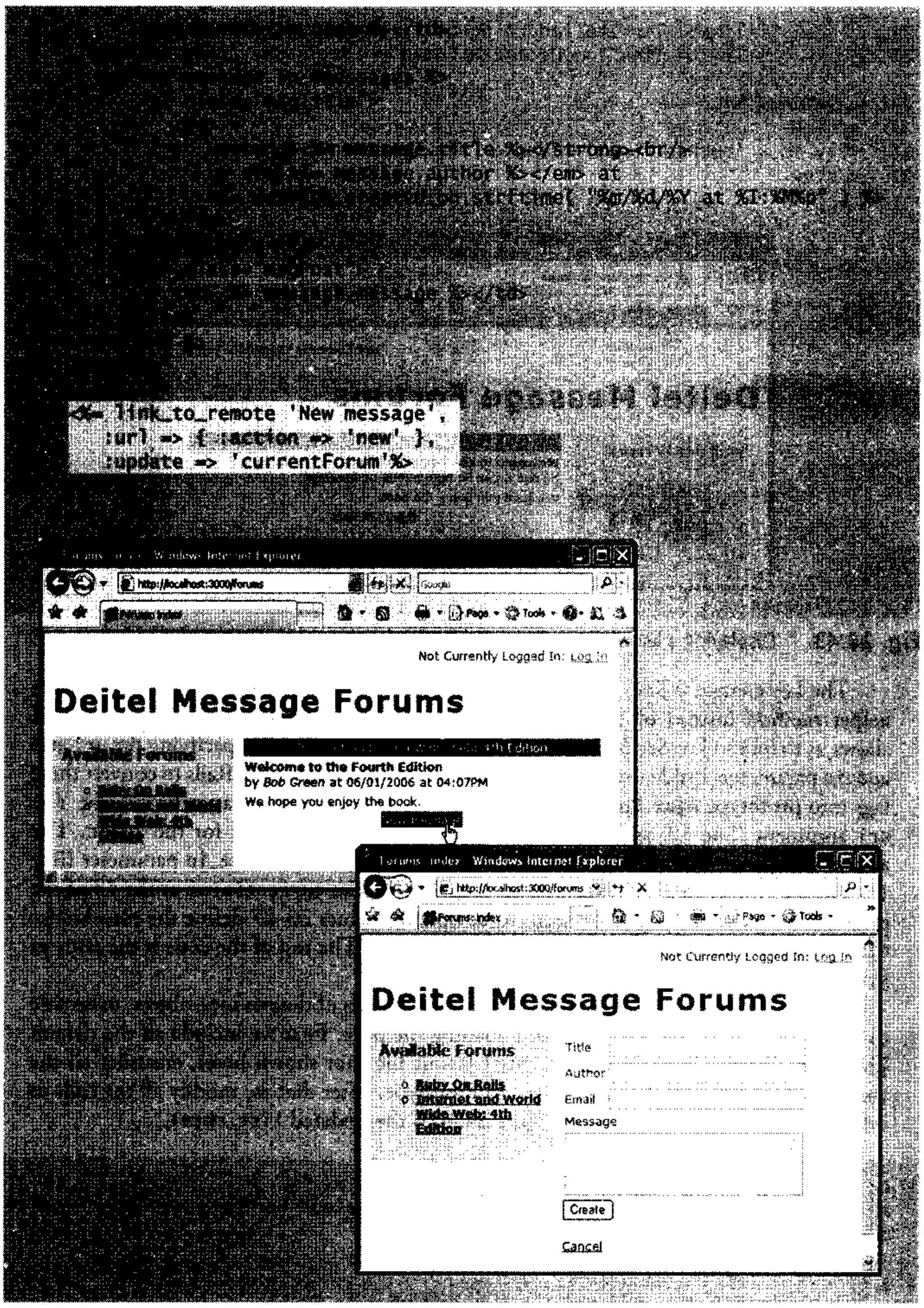
Fig. 24.43 | Displaying a list of messages without reloading the page. (Part 2 of 2.)

The key change is lines 9–12, which have been changed to call the `link_to_remote` helper method instead of the `link_to` helper method. The `link_to_remote` method allows us to link to JavaScript that we included in the layout file. By specifying the `url` and `update` parameters inside the `link_to_remote` method we are telling Rails to convert these tags into **prototype** **`Ajax.Updater`** objects that will update the page asynchronously. The `url` argument (line 10) specifies the `controller` in which to look for the action. The `action` parameter (line 11) specifies the action to invoke. The `forum_id` parameter (line 11) specifies the id to pass to the action. Line 12 specifies `currentForum` as the `id` of the placeholder `div` in the page that needs to be updated. Lines 26–27 define the placeholder `div` element where the list of messages will be inserted. The rest of the code is the same as in the non-Ajax version of this application.

In similar fashion, we modify the `list` and new views of the `message` object, to be able to add a message to a forum without reloading the page. First we include all the default JavaScript libraries in the `message.rhtml` layout file (not shown here), ensuring all the views in the message object have access to Prototype. After that we modify all the calls to other actions to be asynchronous. Figure 24.44 is the updated `list.rhtml`.

```
<!-- Fig. 24.44: app/views/messages/list.rhtml -->
<!-- Forum that allows the user to add a message on the same page. -->
<div class = "messageList">
<table style = "width: 400">
  <tr class="msgHeader">
```

Fig. 24.44 | Forum that allows the user to add a message on the same page. (Part 1 of 2.)



```
<%= link_to_remote 'New message',
   :url => { :action => 'new' },
   :update => 'currentForum'%>
```

Not Currently Logged In: Log In

Deitel Message Forums

Available Forums

- Ruby On Rails
- Internet and World Wide Web: 4th Edition

Welcome to the Fourth Edition
by *Bob Green* at 06/01/2006 at 04:07PM
We hope you enjoy the book.

Not Currently Logged In: Log In

Deitel Message Forums

Available Forums

- Ruby On Rails
- Internet and World Wide Web: 4th Edition

Title
Author
Email
Message

Create

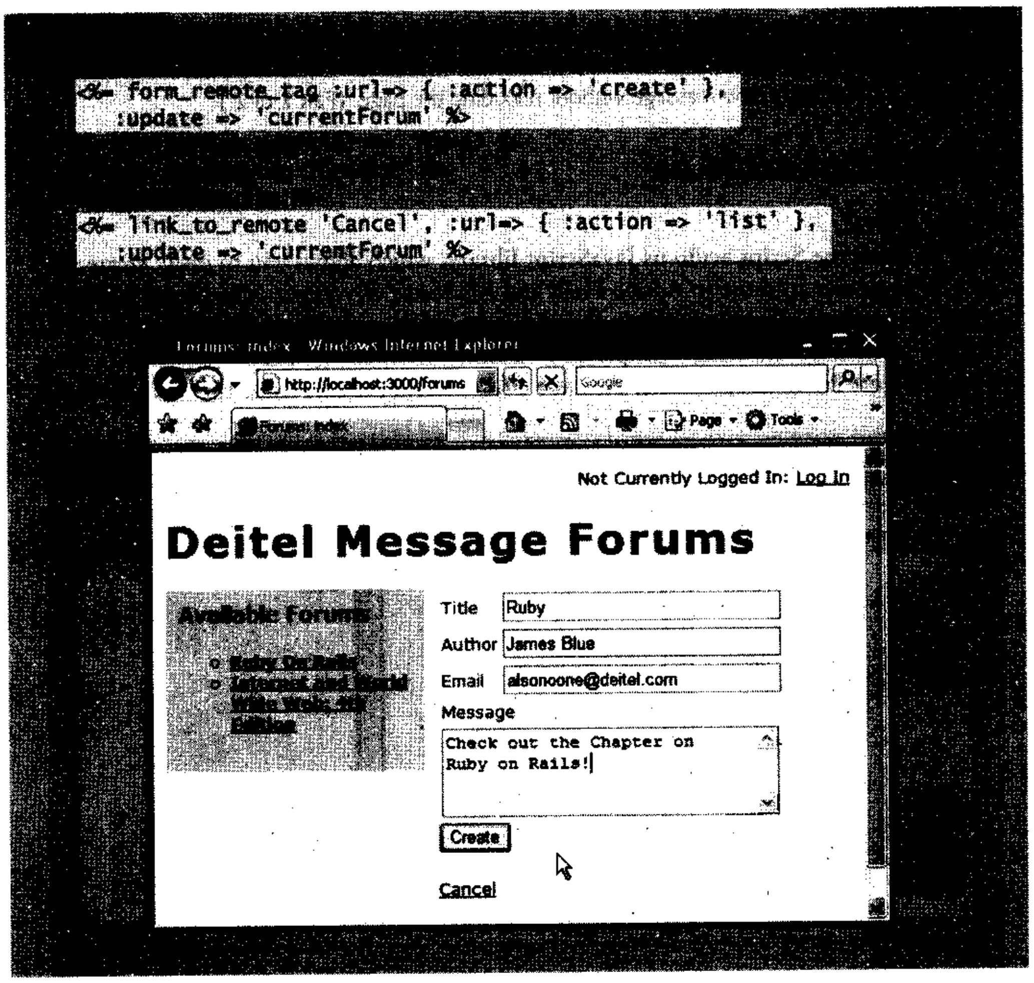
Cancel

Forums: Index - Windows Internet Explorer
http://localhost:3000/forums
Page
Tools

Fig. 24.44 | Forum that allows the user to add a message on the same page. (Part 2 of 2.)

Lines 22–24 use the `link_to_remote` helper method to allow the user to add new messages without reloading the page. The `url` is the **new** action, which returns the form and the placeholder to update is `currentForum`, defined in the `list.rhtml` view of the `forum` object (Fig. 24.43). The new view is also modified, so that once the user submits the new message, the updated `div` named `currentForum` is shown without reloading the page. Figure 24.45 shows the modified `new.rhtml`.

Lines 3–7 have been changed to use the **`form_remote_tag`** helper method, which redirects the client to the next action without reloading the page. Once the user clicks the **Submit** button, generated by `submit_tag` (line 6), the form will generate a Prototype `Ajax.Updater` object that will send the data to the action specified and display the result in the specified placeholder. This placeholder is set to `currentForum`, the same element inside which this forum will be displayed. When the user finishes adding the new message, a new forum will replace this form, without reloading the page. Lines 8–9 provide the user a way to cancel the new-message operation, in which case the original forum displays.



```
<%= form_remote_tag :url=> { :action => 'create' },
   :update => 'currentForum' %>
```

```
<%= link_to_remote 'Cancel', :url=> { :action => 'list' },
   :update => 'currentForum' %>
```

Forums: Index - Windows Internet Explorer
http://localhost:3000/forums
Google
Forums: Index
Page
Tools

Not Currently Logged In: Log In

Deitel Message Forums

Available Forums

- Ruby On Rails
- Internet And World Wide Web 4th Edition

Title	Ruby
Author	James Blue
Email	alsonoone@deitel.com

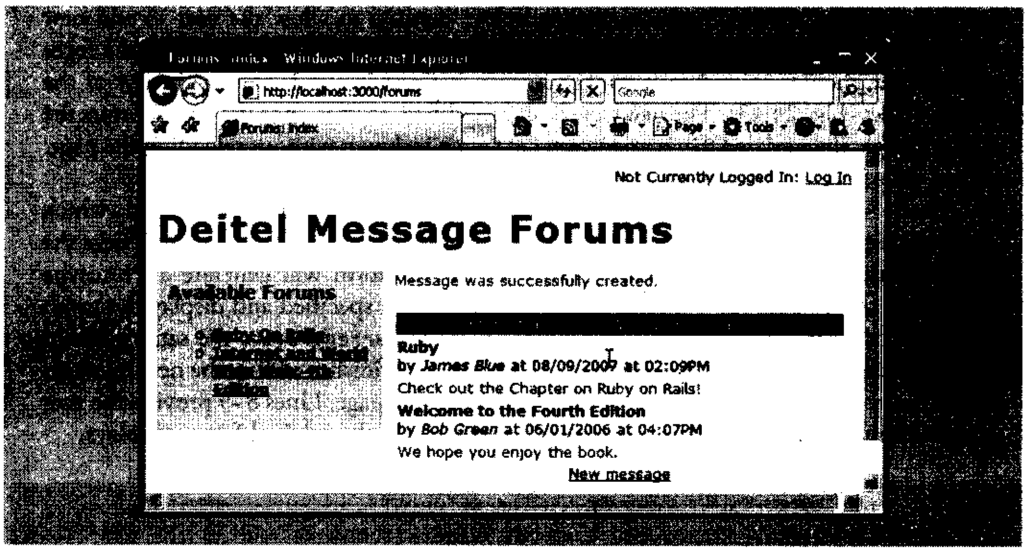
Message

Check out the Chapter on
Ruby on Rails!

Create

Cancel

Fig. 24.45 | Adding a new message without reloading the page. (Part 1 of 2.)



Not Currently Logged In: Log In

Deitel Message Forums

Message was successfully created.

Ruby
by James Blue at 08/09/2007 at 02:09PM
Check out the Chapter on Ruby on Rails!
Welcome to the Fourth Edition
by Bob Green at 06/01/2006 at 04:07PM
We hope you enjoy the book.
New message

Fig. 24.45 | Adding a new message without reloading the page. (Part 2 of 2.)

24.7 Script.aculo.us

Visual Effects

Rails includes the Script.aculo.us JavaScript library, which allows you to easily create visual effects similar to those in Adobe Flash and Microsoft Silverlight. The library provides many pre-defined effects, as well as the ability to create your own effects from the predefined ones. The following example demonstrates many of the effects provided by this library. Figure 24.46 demonstrates the Fade effect. When the user clicks the link above the

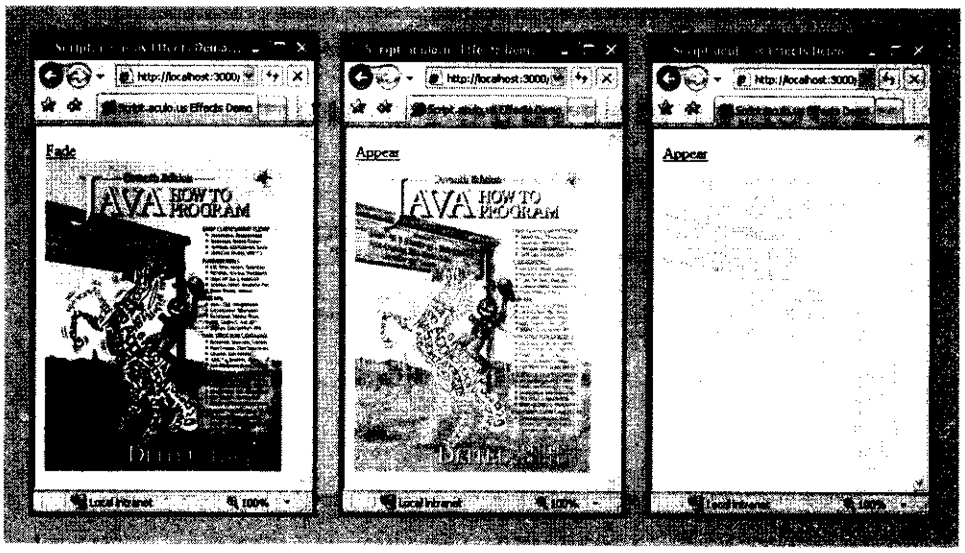


Fig. 24.46 | Script.aculo.us's Fade effect.

image, the effect named in the link will be applied to the image. Once you start the application with Mongrel, open `http://localhost:3000/scriptaculous_demo/` in your web browser.

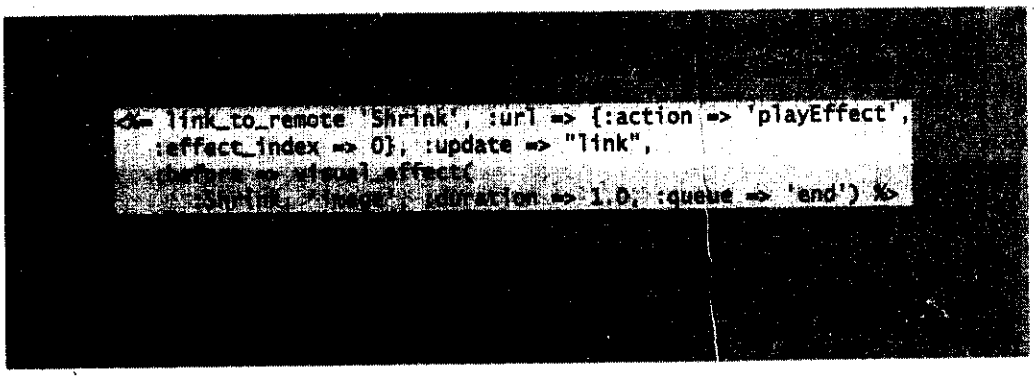
To create this application, first type `rails scriptaculous_demo` in the Ruby console. Next, create the controller by executing

```
ruby script/generate controller ScriptaculousDemo
```

In `app/controllers/scriptaculous_demo_controller.rb` (Fig. 24.47), add the `index` method. This method sets to `0` the `currentEffect` instance variable, which keeps track of which effect the application is currently playing. Next, add the `PlayEffect` method (lines 4–6), which will be called when the user clicks to show the next effect.

Now, create `application.rhtml` (Fig. 24.48) in `app/view/layouts`. This acts as the default layout. Content from `render :partial` commands replaces line 13.

Next, create `index.rhtml` (Fig. 24.49) in `app/views/scriptaculous_demo`. This is the application's default view. The `"link"` `div` (lines 3–8) contains a `link_to_remote` (lines 4–7) that initially is labeled `'Shrink'`, calls `playEffect` with a `effect_index` parameter of `0`, updates itself and plays an effect on the `before` event. The effect is created using the `visual_effect` method (lines 6–7). The parameters of this method call are the effect name, the name of the element the effect should apply to, the `duration` and the



```
<%= link_to_remote 'Shrink', :url => {:action => 'playEffect',
   :effect_index => 0}, :update => "link",
   :before => visual_effect(
   :Shrink, 'image', :duration => 1.0, :queue => 'end') %>
```

Fig. 24.47 | Default view for Script.aculo.us demo.

Fig. 24.48 | Script.aculo.us Demo controller.

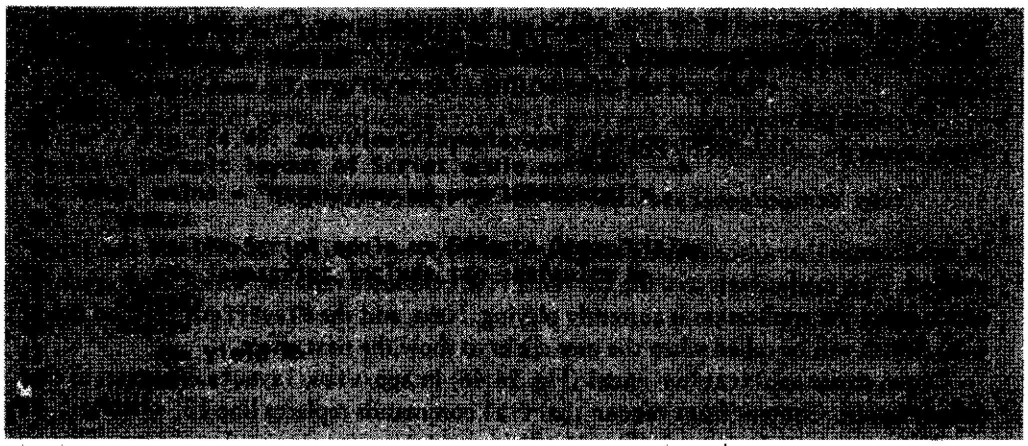


Fig. 24.49 | Default layout of Script.aculo.us demo.

location in the queue. The queue is set to end so that any new effects will be played after all the others are complete. The image in line 11 must be in the public/images directory.

The playEffect method (lines 7–10, Fig. 24.47) sets the currentEffect instance variable to the effect_index parameter, then renders the link view in the link div. In app/views/scriptaculous_demo/_link.rhtml (Fig. 24.50), the application demonstrates several Script.aculo.us effects by using nested if statements to check the currentEffect, apply it, then increment currentEffect after each effect with the effect_index parameter. The link text corresponds to the name of the effect the link activates.

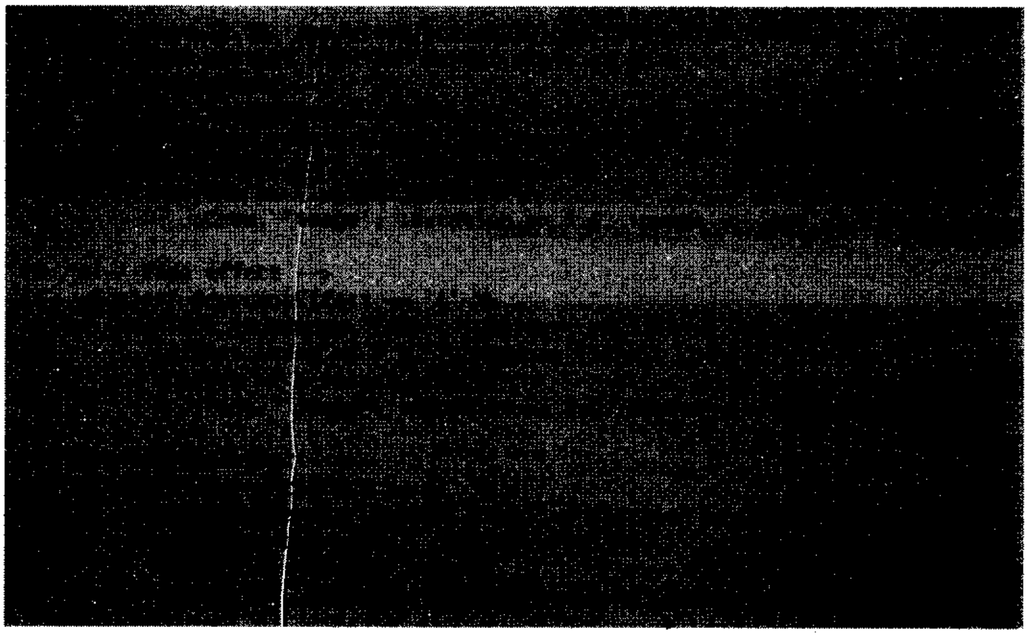


Fig. 24.50 | link partial view for Script.aculo.us demo. (Part 1 of 3.)

```

<!-- BlindUp effect -->
<% elsif @currentEffect == '3' %>
   <%= link_to_remote 'BlindUp', :url => [illegible]
      :effect_index => 4 }, :update => [illegible]
      :before => ( visual_effect(
         :BlindUp, 'image', :duration => [illegible]

<!-- BlindDown effect -->
<% elsif @currentEffect == '4' %>
   <%= link_to_remote 'BlindDown', :url => [illegible]
      :effect_index => 5 }, :update => [illegible]
      :before => ( visual_effect(
         :BlindDown, 'image', :duration => [illegible]

<!-- Puff effect -->
<% elsif @currentEffect == '5' %>
   <%= link_to_remote 'Puff', :url => [illegible]
      :effect_index => 6 }, :update => [illegible]
      :before => ( visual_effect(
         :Puff, 'image', :duration => [illegible]

<!-- SwitchOff effect -->
<% elsif @currentEffect == '6' %>
   <%= link_to_remote 'SwitchOff', :url => [illegible]
      :effect_index => 7 }, :update => [illegible]
      :before => ( visual_effect(
         :SwitchOff, 'image', :duration => [illegible]

<!-- SlideUp effect -->
<% elsif @currentEffect == '7' %>
   <%= link_to_remote 'SlideUp', :url => [illegible]
      :effect_index => 8 }, :update => [illegible]
      :before => ( visual_effect(
         :SlideUp, 'image', :duration => [illegible]

<!-- SlideDown effect -->
<% elsif @currentEffect == '8' %>
   <%= link_to_remote 'SlideDown', [illegible]
      :effect_index => 9 }, [illegible]
      :before => ( visual_effect( [illegible]
         :SlideDown, 'image', [illegible]

<!-- Shake effect -->
<% elsif @currentEffect == '9' %>
   <%= link_to_remote 'Shake', [illegible]
      :effect_index => 10 }, [illegible]
      :before => ( visual_effect( [illegible]
         :Shake, 'image', :duration [illegible]

<!-- Pulsate effect -->
<% elsif @currentEffect == '10' %>
   <%= link_to_remote 'Pulsate', [illegible]
```

Fig. 24.50 | link partial view for Script.aculo.us demo. (Part 2 of 3.)

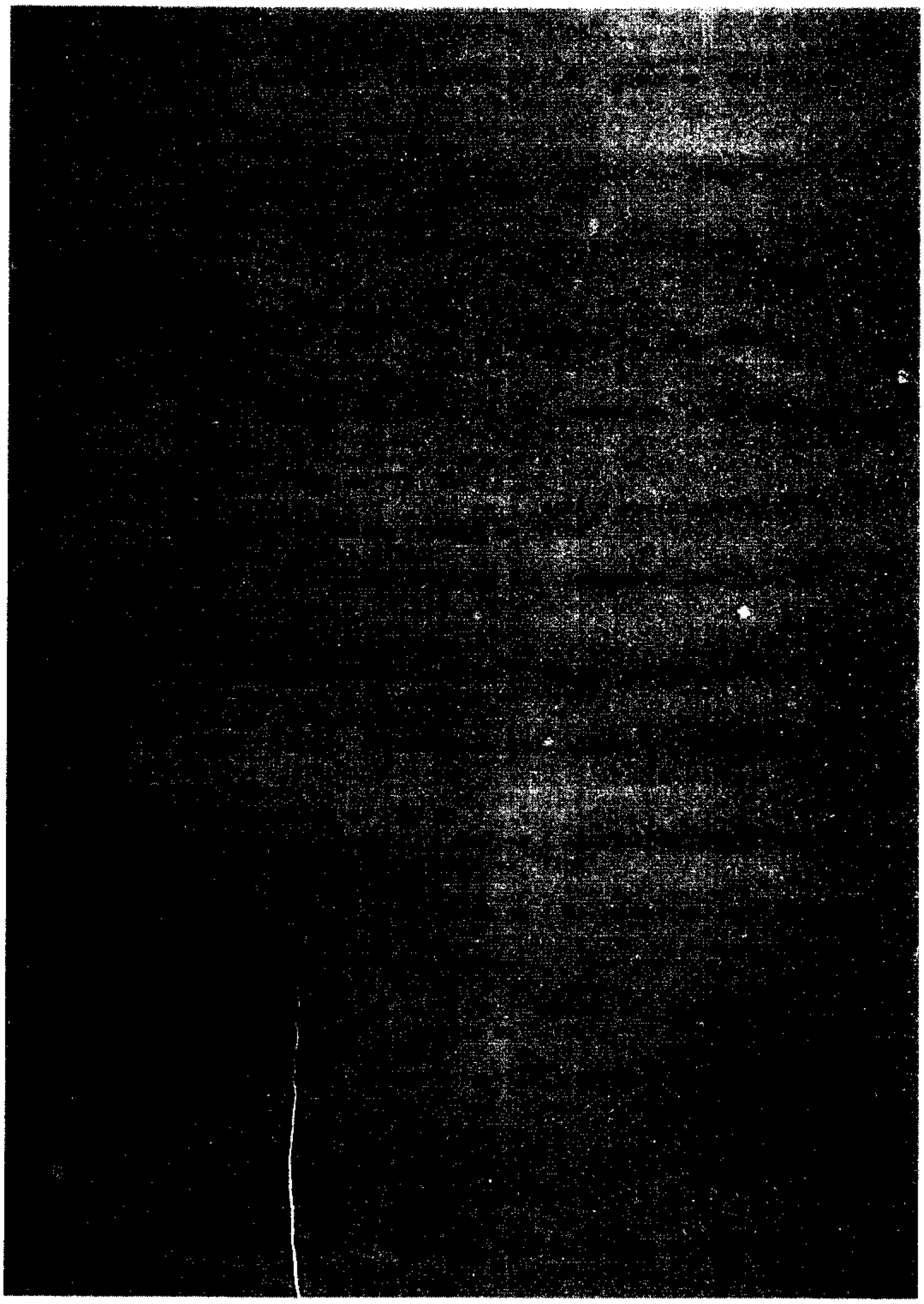


Fig. 24.50 | `link` partial view for Script.aculo.us demo. (Part 3 of 3.)

Other Script.aculo.us Features

The Script.aculo.us library also brings other features to Rails. It provides drag-and-drop capability through the `draggable_element` and `drop_receiving_element` methods. A live example of this can be found at `demo.script.aculo.us/shop`.

Script.aculo.us also provides the `sortable_element` method which allows you to describe a list that allows the user to drag and drop list items to reorder them. A live example of this can be found at `demo.script.aculo.us/ajax/sortable_elements`.

Another intereresting capability is the `text_field_with_auto_complete` method, which enables server-side auto completion of a text field. A live example of this can be found at `demo.script.aculo.us/ajax/autocompleter`.

Flickr Photo Viewer with Effects

The Script.aculo.us library's effects are useful for adding a desktop-like feel to a web page. In the following example (Fig. 24.51), the user can search for photos with specific tags and can specify the number of images for each search to return. The application uses the Script.aculo.us sliding effect to show when the thumbnails for the specified tags have finished loading from Flickr. The application also uses the grow effect when the user clicks an image to display the full-size version of the image.

After creating the `FlickrPhotoViewer` application, you must install the Flickr library for Ruby. This library can be installed by executing `gem install flickr` in the Ruby console. More information about this library is available at `redgreenblu.com/flickr/`. Once

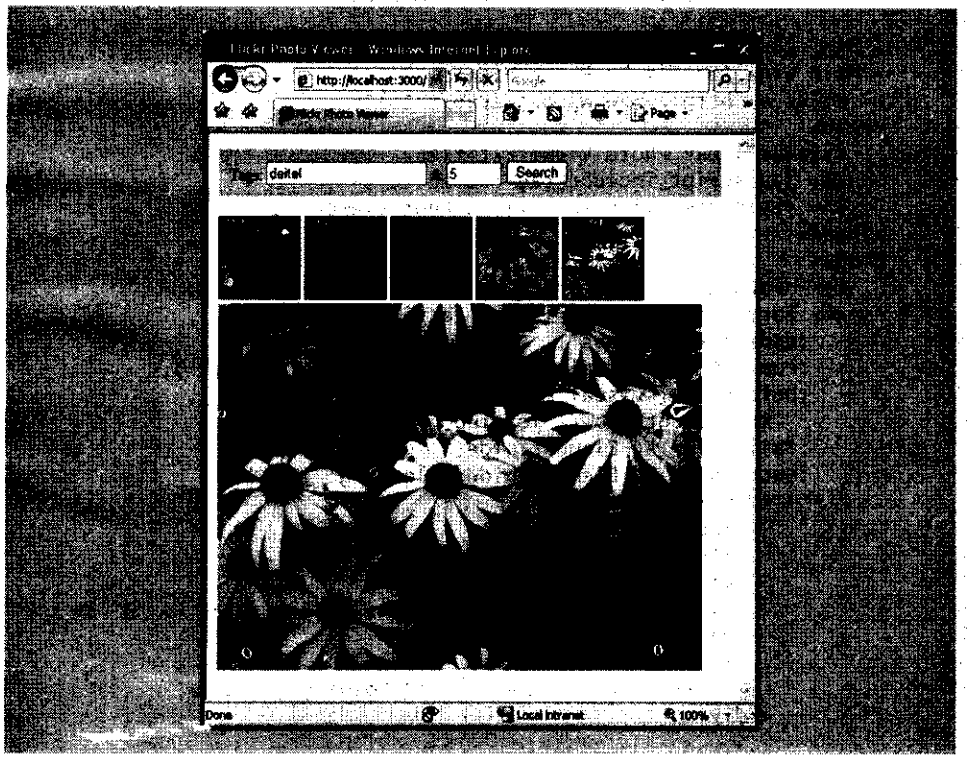


Fig. 24.51 | Flickr Photo Viewer showing search results for **bugs**.

installed, you must configure the library to use your own Flickr API key. You can sign up for a free API key at `www.flickr.com/services/api/misc.api_keys.html`. Once you receive your API key, you *must* replace the key in `flickr.rb` with your own. If you are using Instant Rails, `flickr.rb` will be located in the Instant Rails directory, in the folder If you are running Mac OS X, or otherwise have installed Ruby system-wide, this file will be harder to find. If you cannot locate it with a normal search in Mac OS X, open **Terminal** and use `find / -name flickr.rb` to locate it. The API key to replace should be located at line 57, in the `initialize` method. Finally, you must tell the application to include the Flickr library by adding `require 'flickr'` to the end of `config/environment.rb`.

Create the controller with `ruby script/generate controller flickr`. In `app/views/flickr/index.rhtml` (Fig. 24.52), we create the application's main view. Be sure to copy the `flickrPhotoViewer.css` file from this chapter's folder into the `public/stylesheets/` directory. Lines 15–31 contain a `form_remote_tag` element that implements the application's photo tag search functionality. Line 17 creates a `BlindDown` `visual_effect` for the `thumbs` `div` (line 32) when the `search` `action` is complete. Lines 18–19 create the corresponding `BlindUp` `visual_effect` for the `loading` `div` (lines 28–29). Lines 20–21 hide the `loading` `div` on `failure` and `success` events, respectively. The `fullsizeImage` `div` (line 33) will be populated later with an `img` element.

```
 1  <?xml version = "1.0" [illegible]
 2  <!DOCTYPE html [illegible]
 3     "http://www.w3.org/TR/[illegible]
 4
 5  <!-- Fig. 24.52[illegible]
 6  <!-- Main view for Flickr [illegible]
 7  <html xmlns = "http://[illegible]
 8     <head>
 9        <title>[illegible]
10        <%= javascript[illegible]
11        <%= stylesheet_link_tag[illegible]
12     </head>
13     <body>
14        <!-- Form to search [illegible]
15        <%= form_remote_tag :url => { :action => 'search' },
16           :update => 'thumbs',
17           :complete => visual_effect( :BlindDown, 'thumbs' ),
18           :before => { visual_effect( :BlindUp, 'thumbs' ),
19              %( Element.show( 'loading' ) ) },
20           :failure => %( Element.hide( 'loading' ) ),
21           :success => %( Element.hide( 'loading' ) ) %>
22           <div [illegible]
23              Tag[illegible]
24              <%[illegible]
25              #[illegible]
26              <%[illegible] => "y" %>
27              <%[illegible]
28              <div id = "loading"
29                 style = "display: none">Loading...</div>
30           </div>
```

Fig. 24.52 | Main view for Flick Photo Viewer. (Part 1 of 2.)

```
31       <%= end_form_tag %>
32       <div id = "thumbs"></div>
33       <div id = "fullsizeImage"></div>
34    </body>
35 </html>
```

Fig. 24.52 | Main view for Flickr Photo Viewer. (Part 2 of 2.)

The controller located at `app/controllers/flickr_controller.rb` (Fig. 24.53) handles the `search` `action` called by the form in line 15 of Fig. 24.52 and the `fullsizeImage` action called by the `link_to_remote` in lines 3–9 of Fig. 24.54. In the `search` method, line 6 creates the `flickr` object using the `Flickr` class we installed previously. Lines 7–9 use the `flickr` object to populate `thumbs` with `photos`, supplying as arguments the `tags` and `numImages` values from the corresponding `text_field_tags` in lines 24 and 26 of Fig. 24.52. The `fullsizeImage` method (lines 13–15) takes the `imageURL` parameter's value and uses it to set the `currentURL` variable.

The `thumbs` view (Fig. 24.54) defines each thumbnail as a `link_to_remote` with an `image_tag` as the link's contents. The source of the image is retrieved from the `thumbs` collection that was passed by line 8 of Fig. 24.53. The first index, 0, specifies the image size to be the smallest provided by Flickr. In lines 5–6 of Fig. 24.54, we specify that the `url` should

```
1  # Fig. 24.53: app/controllers/flickr_controller.rb
2  # Controller for Flickr Photo Viewer.
3  class FlickrController < ApplicationController
4     # handle the search request
5     def search
6        flickr = Flickr.new
7        render :partial => "thumbs",
8           :collection => flickr.photos( :tags => params[ :tags ],
9           :per_page => params[ :numImages ] )
10    end # method search
11
12    # handle the thumbnail click, sets the currentURL variable
13    def fullsizeImage
14       @currentURL = params[ :imageURL ]
15    end # method fullsizeImage
16 end # class FlickrController
```

Fig. 24.53 | Controller for Flickr Photo Viewer.

```
1  <!-- Fig. 24.54: app/views/flickr/_thumbs.rhtml -->
2  <!-- thumbs view of Flickr Photo Viewer. -->
3  <%= link_to_remote image_tag( thumbs.sizes[ 0 ][ 'source' ],
4        :class => "image" ),
5     :url => { :action => 'fullsizeImage',
6        :imageURL => thumbs.sizes[ 3 ][ 'source' ] },
7     :update => "fullsizeImage",
8     :success => visual_effect( :grow, 'fullsizeImage',
9        :queue => 'last' ) %>
```

Fig. 24.54 | `thumbs` view of Flickr photo viewer.

activate the `fullsizeImage` action and pass an `imageURL` parameter. This parameter is set to the source of the image's large version. Lines 8–9 apply the `grow` `visual_effect` to `fullsizeImage`.

The `fullsizeImage` view (Fig. 24.55) fills the `fullsizeImage` `div` in line 33 of Fig. 24.52 with an `image_tag`. The source of this image is set to the `currentURL` variable. Try the program out with different tag searches and numbers of images.

```
[illegible]
[illegible]
<%= image_tag( @currentURL, :class => "image" ) %>
```

Fig. 24.55 | `fullsizeImage` view of Flickr Photo Viewer.

24.8 Web Resources

www.deitel.com/Ruby/
www.deitel.com/RubyOnRails/
The Deitel Ruby and Ruby on Rails Resource Centers contain links to some of the best Ruby and Rails resources on the web. There you'll find categorized links to forums, conferences, blogs, books, open source projects, videos, podcasts, webcasts and more. Also check out the tutorials for all skill levels, from introductory to advanced.

Summary

Section 24.1 Introduction

- Ruby on Rails (also known as RoR or just Rails) is a framework for developing data-driven web applications.
- A web framework is a set of libraries and useful tool that can be used to build dynamic web applications.
- Ruby on Rails is different from most other programming languages because it takes advantage of many conventions to reduce development time. If you follow these conventions, the Rails framework generates substantial functionality and perform many tasks for you.
- Ruby on Rails has built-in libraries for performing common web development tasks, such as interacting with a database, sending mass e-mails to clients or generating web services.
- Rails has built-in libraries that provide Ajax functionality, improving the user experience. Rails is quickly becoming a popular environment for web development.
- Ruby on Rails was created by David Heinemeier Hansson of the company 37Signals.

Section 24.2 Ruby

- The Ruby scripting language was developed by Yukihiro "Matz" Matsumoto in 1995 to be a flexible, object-oriented scripting language.
- Ruby's syntax and conventions are intuitive—they attempt to mimic the way a developer thinks. Ruby is an interpreted language.
- Instant Rails is a stand-alone Rails development and testing environment that includes Ruby, Rails, MySQL, Apache, PHP and other components necessary to create and run Rails applications.

- If you are using Mac OS X, there is an application similar to Instant Rails called Locomotive.
- The method `puts` prints the text to the terminal, followed by a newline.
- A method can have parentheses surrounding its parameters, but this is not typical in Ruby unless they are used to avoid ambiguity.
- A line of Ruby code does not have to end with a semicolon, although one can be placed there.
- One way to run a Ruby script is to use the Ruby interpreter.
- IRB (Interactive Ruby) can be used to interpret Ruby code statement by statement.
- Ruby uses dynamic typing, which allows changes to a variable's type at execution time.
- Everything is an object in Ruby, so you can call methods on any piece of data.
- `Hash` `Object`s are mapped to other `Object`s in key/value pairs.
- The exclamation point after a method name is a Ruby convention indicating that the object on which the method is called will be modified.
- Ruby has support for code blocks—groupings of Ruby statements that can be passed to a method as an argument.
- The `initialize` method acts like a constructor in other object-oriented languages—it is used to declare and initialize an object's data.
- When each instance of a class maintains its own copy of a variable, the variable is known as an instance variable and is declared in Ruby using the `@` symbol.
- Classes can also have class variables, declared using the `@@` symbol, that are shared by all copies of a class.
- When an object is concatenated with a string, the object's to_s method is called to convert the object to its string representation.

Section 24.3 Rails Framework

- While users have benefited from the rise of database-driven web applications, web developers have had to implement rich functionality with technology that was not designed for this purpose.
- The Rails framework combines the simplicity of development that has become associated with Ruby with the ability to rapidly develop database-driven web applications.
- Ruby on Rails is built on the philosophy of convention over configuration—if you follow certain programming idioms, your applications will require little or no configuration and Rails will generate substantial portions of the applications for you.
- The Model-View-Controller (MVC) architectural pattern separates application data (contained in the model) from graphical presentation components (the view) and input-processing logic (the controller).
- `ActiveRecord` is used to map a database table to an object.
- `ActionView` is a set of helper methods to modify user interfaces.
- `ActionController` is a set of helper methods to create controllers.

Section 24.4 `ActionController` and `ActionView`

- Ruby on Rails has two classes, `ActionController` and `ActionView`, that work together to process a client request and render a view.
- To generate a controller in Rails, you can use the built-in `Controller` generator by typing `ruby script/generate controller` *`name`*.
- A Ruby on Rails application must be run from a web server.

- Instant Rails comes with a built-in web server named Mongrel, which is easy to use to test Rails applications on the local machine.
- When generating output, a controller usually renders a template—an XHTML document with embedded Ruby that has the `.rhtml` filename extension.
- The `request` object contains the environment variables and other information for a web page.
- Erb (embedded Ruby) that is located between the `<%= %>` tags in rhtml files is parsed as Ruby code and formatted as text.
- A set of Ruby tags without an equals sign—`<% %>`—represents statements to execute as Ruby code but not formatted as text.
- Rails allows you to add headers and footers with a layout—a master view that is displayed by every method in a controller.
- A layout can generate a template for a specific method using `yield`.

Section 24.5 A Database-Driven Web Application

- Rails makes extensive use of Object-Relational Mapping (ORM) that maps a database to application objects.
- The objects that Rails uses to encapsulate a database inherit from `ActiveRecord`.
- One `ActiveRecord` convention is that every model that extends `ActiveRecord::Base` in an application represents a table in a database.
- By convention, the table that the model represents has a name which is the lowercase, pluralized form of the model's name.
- Rails uses a generator to create the `Employee` model. You use a generator by typing `ruby script/server model employee` in the **Ruby Console**, after navigating to your application directory.
- The `ActiveRecord` object has a special feature called `Migration`, which allows you to perform database operations within Rails.
- `ActiveRecord` has built-in functionality for many create, retrieve, update and destroy methods known in Rails as CRUD.
- We can execute the migration using Ruby's `rake` command by typing in `rake db:migrate`, which will call the `self.up` method of all the migrations located in your `db/migrate` directory.
- If you ever want to roll back the migrations, you can type in `rake db:migrate VERSION=0`, which calls each migration's `self.down` method.
- The scaffold method is a powerful tool that automatically creates CRUD functionality. It creates methods such as `new`, `edit` and `list` so you don't have to create them yourself.

Section 24.6 Case Study: Message Forum

- Validators that will be called when the database is modified, can be applied to an object that inherits from `ActiveRecord`.
- The method `validates_presence_of` ensures that all the fields specified by its parameters are not empty.
- The method `validates_format_of` matches all the fields specified by its parameters with a regular expression.
- The `link_to` method is used to link to an action in the controller and pass arguments to it.
- A partial is a block of HTML and embedded Ruby code stored in another file and inserted directly into the document.

- Rails includes a JavaScript library called Prototype that contains easy-to-use cross-browser Ajax functions.
- The `javascript_include_tag` helper method is used to link in JavaScript libraries.
- The `link_to_remote` method allows us to link to JavaScript that we included in the layout file.
- Specifying the `url` and `update` parameters inside the `link_to_remote` method tells Rails to convert these tags into prototype `Ajax.Updater` objects that will update the page asynchronously.

Section 24.7 Script.aculo.us

- Script.aculo.us also provides the `text_field_with_auto_complete` method, which enables server-side autocompletion of a text field.

Terminology

`ActionController`
`ActionView`
`ActiveRecord`
Ajax
Apache
`ApplicationController`
arrays
association
`before_create`
`before_destroy`
`belongs_to`
class variable
code block
comments
controller generator
Convention over Configuration
CRUD
`def`
Don't Repeat Yourself (DRY)
`draggable_element` method
`drop_recieving_element` method
dynamic typing
embedded Ruby (erb)
end
`Errors` Object
escape sequence
`find_all`
`Fixnum`
Gem
`Hash`
initialize
instance variable
Instant Rails
IRB
layout
`link_to` method
`link_to_remote` method
message forum
Model-View-Controller
Mongrel
MySQL
Object Relational Mapping
partial
`password_field` method
PHP
Prototype JavaScript Library
`puts`
relational integrity
`reset_session`
request object
Rails
RoR
Ruby
Ruby interpreter
Ruby on Rails
scaffold
`scaffold` generator
Script.aculo.us JavaScript library
session variable
`sortable_element` method
`String`
template
`text_field` method
`text_field_with_autocomplete` method
`to_s` method
`validate`
`validates_format_of`
`validates_presence_of`
validations
verify
web framework
web server
web services

Self-Review Exercises

24.1 Fill in the blanks in each of the following statements:
a) ________ is a stand-alone Rails development and testing environment for Windows.
b) Ruby on Rails is built on the philosophy of ________.
c) The ________ architectural pattern separates application data from graphical presentation components and input processing.
d) The objects that Rails uses to encapsulate a database inherit from class ________.
e) A(n) ________ is a master view that is displayed by every method in a controller.
f) The ________ method automatically creates CRUD functionality in Ruby on Rails.
g) The ________ helper method creates a link in HTML that calls a partial page update.
h) The ________ helper method calls effects from the Script.aculo.us library.

24.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Every line in Ruby must end with a semicolon.
b) Rails is a programming language.
c) Rails makes creating database-driven Internet applications easy.
d) The name of the model must be the same as the name of the table that is associated with it.
e) By following Ruby on Rails naming conventions you can build a Rails application with no configuration.
f) Each controller in Ruby has one layout file which is rendered for every action of that controller.
g) Embedded Ruby located between the <% %> delimiters is evaluated and rendered on the page.
h) Rails implements Ajax functionality using the JavaScript Prototype library.

Exercises

24.3 Write a series of `ActiveRecord::Migration` scripts to set up the structure for a book catalog database. The book catalog should have two tables—`Books` and `Authors`. The `Authors` table should have fields containing the author ID (the primary key for the table) the first name and the last name. The `Books` table should have fields containing the book's id (primary key for the table) the title, the author's ID that corresponds to an id in the authors table, the most current edition number and the year the most current edition was released. The migration scripts should also add a few rows of data to both tables.

24.4 Assume you have a book catalog database with tables `Books` and `Authors`. Write down the Ruby commands you would put in the **Command Prompt** to set up a structure for a Book Catalog application. Do not create your own directories or files—let the scripts do that for you. Both `Books` and `Authors` should have a model, a view and a controller component associated with them. There should be a structure set up for CRUD functionality associated with them.

24.5 Modify the Forum case study to be completely Ajax enabled. Make the Create Forum and Delete Forum links open up on the same page instead of linking to a different page. You will have to modify the `list.rhtml`, `new.rhtml` and `delete.rhtml` files in the `Forum` section of the `View` directory.

24.6 Create an Address Book application like the one in Fig. 15.9. Enable the user to expand and contract, add and remove address-book entries.