



Ruby on Rails 2.1.X



tutorialspoint
SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

Ruby on Rails is an extremely productive web application framework written in Ruby by David Heinemeier Hansson. This tutorial gives you a complete understanding on Ruby on Rails 2.1.

Audience

This tutorial has been designed for beginners who would like to use the Ruby framework for developing database-backed web applications.

Prerequisites

Before you start Ruby on Rails, please make sure you have a basic understanding on the following subjects:

- Ruby syntax and language constructs such as blocks.
- Relational databases and SQL.
- HTML documents and CSS.

Copyright & Disclaimer

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial	
Audience.....	i
Prerequisites.....	i
Copyright & Disclaimer	i
Table of Contents.....	ii
1. INTRODUCTION.....	1
What is Ruby?.....	1
Why Ruby?	1
Sample Ruby Code	2
Embedded Ruby.....	2
What is Rails?	4
Full Stack Framework.....	4
Convention over Configuration	4
Don't Repeat Yourself (DRY)	4
Rails Strengths	4
What is Rails 2.1.0?	5
2. INSTALLATION.....	6
Rails Installation on Windows	6
Rails Installation on Mac OS X.....	7
Rails Installation on Linux	7
Keeping Rails Up-to-Date	8
Installation Verification	9

How to Upgrade?	10
3. FRAMEWORK.....	11
Ruby on Rails MVC Framework	11
Representation of MVC Framework.....	12
Directory Representation of MVC Framework	13
4. DIR STRUCTURE	15
5. EXAMPLES.....	18
Workflow for Creating Rails Applications.....	18
Creating an Empty Rails Application	18
Starting Web Server	19
6. DATABASE SETUP.....	21
Configuring database.yml	21
7. ACTIVE RECORDS	23
Translating a Domain Model into SQL.....	23
Creating Active Record Files.....	23
Creating Associations between Models	24
Implementing Validations.....	25
8. MIGRATIONS.....	26
What can Rails Migration Do?.....	26
Create the Migrations	27
Edit the Code to Tell it What to Do.....	27
Run the Migration.....	29
Running Migrations for Production and Test Databases	29

9. CONTROLLERS.....	31
Implementing the list Method	32
Implementing the show Method.....	32
Implementing the new Method	33
Implementing the create Method	33
Implementing the edit Method.....	34
Implementing the update Method.....	34
Implementing the delete Method	35
Additional Methods to Display Subjects.....	35
10. VIEWS	38
Creating View File for list Method.....	38
Creating View File for new Method.....	40
Creating View File for show Method	42
Creating View File for edit Method	43
Creating View File for delete Method	45
Creating View File for show_subjects Method	45
11. LAYOUTS	48
Adding Style Sheet	50
12. SCAFFOLDING	53
Scaffolding Example	53
Creating an Empty Rails Web Application	53
Setting Up the Database	53
Database Table Definition.....	55
The Generated Scaffold Code.....	55

The Controller	56
The Views	59
The Migrations.....	59
Ready to Test	60
Enhancing the Model	62
How Scaffolding is Different?	63
13. AJAX ON RAILS	64
How Rails Implements Ajax.....	64
AJAX Example	65
Creating Controller	65
Creating Views	67
Adding Ajax Support	68
Creating Partial for create Method	69
14. FILE UPLOADING	71
Creating the Model	71
Creating Controller	72
Creating View	73
Files Uploaded from Internet Explorer	74
Deleting an Existing File	75
15. SENDING EMAILS	76
Setting Up the Database	76
ActionMailer – Configuration	77
Generate a Mailer	78
Creating the Controller	79

Defining Views	80
Rest for Testing	81
Sending HTML Emails using Rails	82
16. RMAGICK	83
Converting Image Formats	84
17. HTTP BASIC AUTHENTICATION	86
18. EXCEPTION HANDLING	89
Where to Log Errors?	90
19. ROUTES SYSTEM	91
Route Priority	92
Modifying the Default Route	92
The Ante-Default Route	93
The Empty Route	93
Named Routes	94
Pretty URLs	94
20. UNIT TESTING	95
Introduction	95
Rails Testing	95
Database Setup	95
Configuring database.yml:	96
Generate Migration	97
Testing Models	98
Testing the Controllers	100
Using Rake for Testing	104

21. QUICK REFERENCE GUIDE	106
Ruby on Rails – Rake Utility	106
Ruby on Rails – The Scripts.....	108
Ruby on Rails – The Plugins	109
Ruby on Rails – The Generators	109
Ruby on Rails – Model Relations	110
Association Join Models.....	112
Ruby on Rails – Controller Methods.....	113
Ruby on Rails – Layouts.....	114
Adding a Stylesheet	116
Ruby on Rails – Render	118
Ruby on Rails – HTML Forms	120
Checkbox Button.....	123
Ruby on Rails – RXML.....	124
Ruby on Rails – RHTML	126
Ruby on Rails – HTML Links	127
Ruby on Rails – Session and Cookies	128
Ruby on Rails – User Input Validations.....	129
Ruby on Rails – Maths Functions.....	131
Ruby on Rails – Finders	132
Ruby on Rails – Nested with-scope	133
Ruby on Rails – Callback Functions.....	135

1. Introduction

What is Ruby?

Before we ride on Rails, let us recapitulate a few points of Ruby, which is the base of Rails.

Ruby is the successful combination of:

- Smalltalk's conceptual elegance,
- Python's ease of use and learning, and
- Perl's pragmatism.

Ruby is

- A high-level programming language.
- Interpreted like Perl, Python, Tcl/Tk.
- Object-oriented like Smalltalk, Eiffel, Ada, Java.

Why Ruby?

Ruby originated in Japan and now it is gaining popularity in US and Europe as well. The following factors contribute towards its popularity:

- Easy to learn
- Open source (very liberal license)
- Rich libraries
- Very easy to extend
- Truly object-oriented
- Less coding with fewer bugs
- Helpful community

Although we have many reasons to use Ruby, there are a few drawbacks as well that you may have to consider before implementing Ruby:

- Performance Issues - Although it rivals Perl and Python, it is still an interpreted language and we cannot compare it with high-level programming languages like C or C++.

- Threading model – Ruby does not use native threads. Ruby threads are simulated in the VM rather than running as native OS threads.

Sample Ruby Code

Here is a sample Ruby code to print "Hello Ruby".

```
#!/usr/bin/ruby -w

# The Hello Class
class Hello
  # Define constructor for the class
  def initialize( name )
    @name = name.capitalize
  end

  # Define a ruby method
  def salute
    puts "Hello #{@name}!"
  end
end

# Create a new object for Hello class
obj = Hello.new("Ruby")

# Call ruby method
obj.salute
```

This will produce the following result:

```
Hello Ruby
```

For a complete understanding on **Ruby**, please go through our **Ruby** Tutorial

Embedded Ruby

Ruby provides a program called ERb (Embedded Ruby), written by *Seki Masatoshi*. ERb allows you to put Ruby code inside an HTML file. ERb reads

along, word for word, and then at a certain point, when it encounters a Ruby code, it starts executing the Ruby code.

You need to know only two things to prepare an ERb document:

- If you want some Ruby code executed, enclose it between **<%** and **%>**.
- If you want the result of the code execution to be printed out, as a part of the output, enclose the code between **<%=** and **%>**.

Here's an example. Save the code in `erbdemo.erb` file. Note that a Ruby file will have an extension **.rb**, while an Embedded Ruby file will have an extension **.erb**.

```
<% page_title = "Demonstration of ERb" %>
<% salutation = "Dear programmer," %>
<html>
<head>
<title><%= page_title %></title>
</head>
<body>
<p><%= salutation %></p>
<p>This is an example of how ERb fills out a template.</p>
</body>
</html>
```

Now, run the program using the command-line utility `erb`.

```
c:\ruby\>erb erbdemo.erb
```

This will produce the following result:

```
<html>
<head>
<title>Demonstration of ERb</title>
</head>
<body>
<p>Dear programmer,</p>
<p>This is an example of how ERb fills out a template.</p>
</body>
</html>
```

What is Rails?

- An extremely productive web-application framework.
- You could develop a web application at least ten times faster with Rails, than you could with a typical Java framework.
- An open source Ruby framework for developing database-backed web applications.
- Your code and database schema are the configuration!
- No compilation phase required.

Full Stack Framework

- Includes everything needed to create a database-driven web application using the Model-View-Controller (MVC) pattern.
- Being a full-stack framework means all the layers are built to work seamlessly with less code.
- Requires fewer lines of code than other frameworks.

Convention over Configuration

- Rails shuns configuration files in favor of conventions, reflection, and dynamic run-time extensions.
- Your application code and your running database already contain everything that Rails needs to know!

Don't Repeat Yourself (DRY)

DRY is a slogan you will hear frequently associated with Ruby on Rails, which means you need to code the behavior only once and you never have to write similar code in two different places. This is important because you are less likely to make mistakes by modifying your code at one place only.

Rails Strengths

Rails is packed with features that make you more productive, with many of the following features building on one other.

Metaprogramming: Other frameworks use extensive code generation from scratch. Metaprogramming techniques use programs to write programs. Ruby is

one of the best languages for metaprogramming, and Rails uses this capability well. Rails also uses code generation but relies much more on metaprogramming for the heavy lifting.

Active Record: Rails introduces the Active Record framework, which saves objects to the database. The Rails version of the Active Record discovers the columns in a database schema and automatically attaches them to your domain objects using metaprogramming.

Convention over configuration: Most web development frameworks for .NET or Java force you to write pages of configuration code. If you follow the suggested naming conventions, Rails doesn't need much configuration.

Scaffolding: You often create temporary code in the early stages of development to help get an application up quickly and see how major components work together. Rails automatically creates much of the scaffolding you'll need.

Ajax at the core: Ajax is the technology that has become a standard to provide interactivity to websites without becoming intrusive. Ruby on Rails has a great support for Ajax technology and it is a part of the core libraries. So, when you install RoR, Ajax support is also made available to you.

Built-in testing: Rails creates simple automated tests you can then extend. Rails also provides supporting code called harnesses and fixtures that make test cases easier to write and run. Ruby can then execute all your automated tests with the rake utility.

Three environments: Rails gives you three default environments: development, testing, and production. Each behaves slightly differently, making your entire software development cycle easier. For example, Rails creates a fresh copy of the Test database for each test run.

What is Rails 2.1.0?

This is the latest version of Ruby on Rails, which has been released by the Rails core team on Saturday May 31, 2008.

This version is a further improvement on RoR 2.0, which was again really a fantastic release, absolutely stuffed with great new features, loads of fixes, and an incredible amount of polish over its previous versions RoR 1.2.x.

This tutorial takes you through all the important features available in the latest RoR version 2.1.0.

After this tutorial, you should be able to build your website using one of the best Web 2.0 technologies called Ruby on Rails v2.1.0.

2. Installation

To develop a web application using Ruby on Rails Framework, you would need to install the following software:

- Ruby
- The Rails framework
- A Web Server
- A Database System

We assume that you already have installed a Web Server and Database System on your computer. You can always use the WEBrick Web Server, which comes with standard installation of Ruby. Most sites, however, use Apache or lightTPD in production.

Rails works with many database systems, including MySQL, PostgreSQL, SQLite, Oracle, DB2 and SQL Server. Please refer to a corresponding Database System Setup manual to setup your database.

Let's look at the installation instructions for Rails' Framework on Windows, Mac OS X, and Linux.

Rails Installation on Windows

First, let's check to see if you already have Ruby installed. Bring up a command prompt and type **C:\> ruby -v**. If Ruby responds, and if it shows a version number at or above 1.8.6, then type **C:\> gem --version**. If you don't get an error, skip to step 3. Otherwise, we'll do a fresh installation for Ruby.

1. If Ruby is not installed, then download an installation package from rubyinstaller.rubyforge.org. Follow the **download** link, and run the resulting installer. This is an exe like **ruby186-25.exe** and will be installed in a single click. You may as well install everything. It's a very small package, and you'll get **RubyGems** as well along with this package.
2. With RubyGems loaded, you can install all of Rails and its dependencies through the command line:

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

The above command may take some time to install all dependencies. Make sure you are connected to the internet while installing gems dependencies.

Congratulations! You are now on Rails over Windows.

NOTE: In case you face any problem with the above installation, there are chances that you may not have the latest versions of Ruby or other Gems. So just issue the following command and you will have everything updated automatically.

```
C:\> gem update
```

Then try the above command with updated gems.

Rails Installation on Mac OS X

1. First, let's check to see if you already have Ruby installed. Bring up a command prompt and type **\$ ruby -v**. If Ruby responds, and if it shows a version number at or above 1.8.6 then skip to step 3. Otherwise, we'll do a fresh installation for Ruby. To install a fresh copy of Ruby, the Unix instructions that follow should help.
2. Next, you have to install RubyGems. Go to rubygems.rubyforge.org and follow the download link. OS X will typically unpack the archive file for you, so all you have to do is navigate to the downloaded directory and (in the Terminal application) type the following:

```
tp> tar xzf rubygems-0.8.10.tar.gz
tp> cd rubygems-0.8.10
rubygems-0.8.10> sudo ruby setup.rb
```

3. Now, use RubyGems to install Rails. Issue the following command.

```
tp> sudo gem install rails --include-dependencies
```

The above command may take some time to install all dependencies. Make sure you are connected to the internet while installing gems dependencies.

Congratulations! You are now on Rails over Mac OS X.

NOTE: In case you face any problem with above installation, there are chances that you may not have the latest versions of Ruby or other Gems. So just issue the following command and you will have everything updated automatically.
tp> sudo gem update

Then try the above command with updated gems.

Rails Installation on Linux

1. First, let's check to see if you already have Ruby installed. Bring up a command prompt and type **\$ ruby -v**. If Ruby responds, and if it shows a

version number at or above 1.8.6, then skip to step 5. Otherwise, we'll do a fresh installation for Ruby.

2. Download ruby-x.y.z.tar.gz from www.ruby-lang.org.
3. Untar the distribution, and enter the top-level directory.
4. Do the usual open-source build as follows:

```
tp> tar -xzf ruby-x.y.z.tar.gz
tp> cd ruby-x.y.z
ruby-x.y.z> ./configure
ruby-x.y.z> make
ruby-x.y.z> make test
ruby-x.y.z> make install
```

5. Install RubyGems. Go to rubygems.rubyforge.org, and follow the **download** link. Once you have the file locally, enter the following at your command prompt:

```
tp> tar -xzf rubygems-x.y.z.tar.gz
tp> cd rubygems-x.y.z
rubygems-x.y.z> ruby setup.rb
```

6. Now use RubyGems to install Rails. Still in the shell, issue the following command.

```
tp> gem install rails --include-dependencies
```

The above command may take some time to install all dependencies. Make sure you are connected to the internet while installing gems dependencies.

Congratulations! You are now on Rails over Linux.

NOTE: In case you face any problem with above installation, there are chances that you may not have the latest versions of Ruby or other Gems. So, just issue the following command and you will have everything updated automatically.
tp> sudo gem update

Then try the above command with updated gems.

Keeping Rails Up-to-Date

Assuming you have installed Rails using RubyGems, keeping it up-to-date is relatively easy. Issue the following command:


```
tp> gem update rails
```

This will automatically update your Rails installation. The next time you restart your application, it will pick up this latest version of Rails. While giving this command, make sure you are connected to the internet.

Installation Verification

You can verify if everything is setup according to your requirements or not. Use the following command to create a *demo project* in Rails environment.

```
tp> rails demo
```

This will create a demo rails' project using **SQLite** database. Note that Rails uses **SQLite** as its default database.

We can create an application that will use **MySQL** database. Assuming you have **MySQL** database setup on your machine, issue the following command to create an application that will use MySQL database:

```
tp> rails -d mysql demo
```

We will discuss the database setup part in subsequent chapters. Currently we have to check if our environment is setup properly or not. Use the following commands to run *WEBrick* web server on your machine:

```
tp> cd demo
demo> ruby script/server
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2007-02-26 09:16:43] INFO WEBrick 1.3.1
[2007-02-26 09:16:43] INFO ruby 1.8.2 (2004-08-24)...
[2007-02-26 09:16:43] INFO WEBrick::HTTPServer-start:pid=2836...
....
```

Now open your browser and type the following address text box.

```
http://localhost:3000
```

You should receive a message like "Welcome aboard" or "Congratulations".

How to Upgrade?

If you are already running an old version of Rails, then here is the procedure to upgrade it to the latest version 2.1:

1. If you want to move your application to Rails 2.0, you should first move it to Rails 1.2.6.
2. If your application runs fine on 1.2.6 with no deprecation warnings, there's a good chance that it'll run straight up on 2.0.
3. To complete the upgrade, you would have to upgrade your extractions. If you are using *pagination*, you will need to install the *classic_pagination* plugin. If you are using *Oracle*, you will need to install the *activerecord-oracle-adapter* gem.

3. Framework

A framework is a program, set of programs, and/or code library that writes most of the applications for you. When you use a framework, your job is to write the parts of the application that make it do the specific things you want.

When you set out to write a Rails application, leaving aside the configuration and other housekeeping chores, you have to perform three primary tasks:

- **Describe and model your application's domain:** The domain is the universe of your application. The domain may be a music store, a university, a dating service, an address book, or a hardware inventory. So, you have to figure out what's in it, what entities exist in this universe, and how the items in it relate to each other. This is equivalent to modeling a database structure to keep the entities and their relationship.
- **Specify what can happen in this domain:** The domain model is static. You have to make it dynamic. Addresses can be added to an address book. Musical scores can be purchased from music stores. Users can log in to a dating service. Students can register for classes at a university. You need to identify all the possible scenarios or actions that the elements of your domain can participate in.
- **Choose and design the publicly available views of the domain:** At this point, you can start thinking in Web-browser terms. Once you've decided that your domain has students, and they can register for classes, you can envision a welcome page, a registration page, or a confirmation page, etc. Each of these pages or views shows the user how things stand at a certain point.

Based on the above three tasks, Ruby on Rails deals with a Model/View/Controller (MVC) framework.

Ruby on Rails MVC Framework

The **M**odel **V**iew **C**ontroller principle divides the work of an application into three separate but closely cooperative subsystems.

Model (ActiveRecord)

Maintains the relationship between Object and Database and handles validation, association, transactions, and more.

This subsystem is implemented in **ActiveRecord** library, which provides an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records.

Ruby method names are automatically generated from the field names of database tables.

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

View (ActionView)

It is a presentation of data in a particular format, triggered by a controller's decision to present the data. They are script-based templating systems like JSP, ASP, PHP and very easy to integrate with AJAX technology.

This subsystem is implemented in **ActionView** library, which is an Embedded Ruby (ERb) based system for defining presentation templates for data presentation. Every Web connection to a Rails application results in the displaying of a view.

ActionView helps in separating the details of presentation from the core business logic of your application.

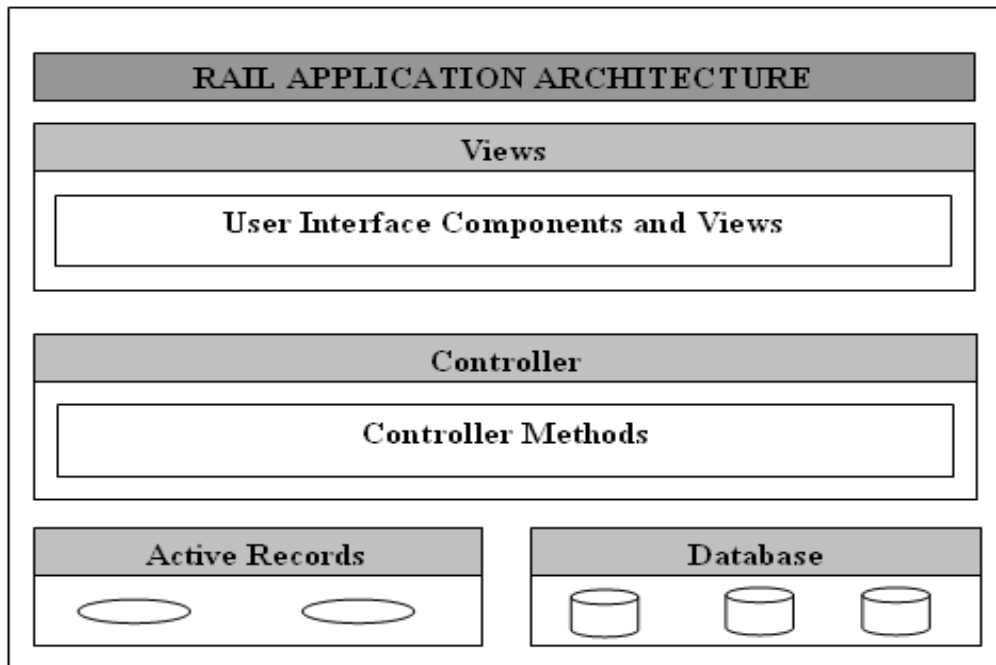
Controller (ActionController)

The facility within the application that directs traffic, on the one hand querying the models for specific data, and on the other hand, organizing that data (searching, sorting, massaging it) into a form that fits the needs of a given view.

This subsystem is implemented in ActionController, which is a data broker sitting between the ActiveRecord (the database interface) and the ActionView (the presentation engine).

Representation of MVC Framework

A pictorial representation of Ruby on Rails Framework is given here:



Directory Representation of MVC Framework

Assuming a standard, default installation over Linux, you can find them like this:

```
tp> cd /usr/local/lib/ruby/gems/1.8/gems
tp> ls
```

You will see subdirectories including (but not limited to) the following:

- actionpack-x.y.z
- activerecord-x.y.z
- rails-x.y.z

Over a Windows installation, you can find them like this:

```
C:\>cd ruby\lib\ruby\gems\1.8\gems
C:\ruby\lib\ruby\gems\1.8\gems>dir
```

You will see subdirectories including (but not limited to) the following:

- actionpack-x.y.z
- activerecord-x.y.z
- rails-x.y.z

ActionView and ActionController are bundled together under ActionPack.

ActiveRecord provides a range of programming techniques and shortcuts for manipulating data from an SQL database. ActionController and ActionView provides facilities for manipulating and displaying that data. Rails ties them all together.

4. Dir Structure

When you use the helper script of Rails to create your application, it creates the entire directory structure for your application. Rails knows where to find things it needs within this structure, so you don't have to provide any input.

Here is a top-level view of the directory tree created by the helper script at the time of an application creation. Except for minor changes between the releases, every Rails project will have the same structure with the same naming conventions. This consistency gives you a tremendous advantage; you can quickly move between Rails projects without re-learning the project's organization.

To understand this directory structure, let's use the **demo** application created in the installation chapter. This can be created using a simple helper command as follows:

```
C:\ruby\> rails -d mysql demo
```

Now, go into the demo application root directory as follows:

```
C:\ruby\> cd demo
C:\ruby\demo> dir
```

You will find a directory structure as follows:

```
demo/
.... /app
..... /controller
..... /helpers
..... /models
..... /views
..... /layouts
.... /config
.... /db
.... /doc
.... /lib
.... /log
.... /public
```

```
.... /script
.... /test
.... /tmp
.... /vendor
README
Rakefile
```

Now let's explain the purpose of each directory.

- **app:** It organizes your application components. It's got subdirectories that hold the view (views and helpers), controller (controllers), and the backend business logic (models).
- **app/controllers:** The controllers subdirectory is where Rails looks to find controller classes. A controller handles a web request from the user.
- **app/helpers:** The helpers subdirectory holds any helper classes used to assist the model, view, and controller classes. It helps to keep the model, view, and controller code small, focused, and uncluttered.
- **app/models:** The models subdirectory holds the classes that model and wrap the data stored in our application's database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it dead simple!
- **app/view:** The views subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user's browser.
- **app/view/layouts:** Holds the template files for layouts to be used with views. This models the common header/footer method of wrapping views. In your views, define a layout using the `<tt>layout :default</tt>` and create a file named `default.rhtml`. Inside `default.erb`, call `<% yield %>` to render the view using this layout.
- **config:** This directory contains a small amount of configuration code that your application will need, including your database configuration (in `database.yml`), your Rails environment structure (`environment.rb`), and routing of incoming web requests (`routes.rb`). You can also tailor the behavior of the three Rails environments for test, development, and deployment with files found in the `environments` directory.
- **db:** Usually, your Rails application will have model objects that access relational database tables. You can manage the relational database with scripts you create and place in this directory.

- **doc:** This directory is where your application documentation will be stored when generated using **rake doc:app**.
- **lib:** Application-specific libraries go here. Basically, any kind of custom code that doesn't belong under controllers, models, or helpers. This directory is in the load path.
- **log:** Error logs go here. Rails creates scripts that help you manage various error logs. You'll find separate logs for the server (server.log) and each Rails environment (development.log, test.log, and production.log).
- **public:** Like the public directory for a web server, this directory has web files that don't change, such as JavaScript files (public/javascripts), graphics (public/images), stylesheets (public/stylesheets), and HTML files (public). This should be set as the DOCUMENT_ROOT of your web server.
- **script:** This directory holds scripts to launch and manage the various tools that you'll use with Rails. For example, there are scripts to generate code (generate) and launch the web server (server), etc.
- **test:** The tests you write and those Rails creates for you, all goes here. You'll see a subdirectory for mocks (mocks), unit tests (unit), fixtures (fixtures), and functional tests (functional).
- **tmp:** Rails uses this directory to hold temporary files for intermediate processing.
- **vendor:** Libraries provided by third-party vendors (such as security libraries or database utilities beyond the basic Rails distribution) goes here.

Apart from these directories, there will be two files available in the demo directory.

- **README:** This file contains a basic detail about Rail Application and description of the directory structure explained above.
- **Rakefile:** This file is similar to Unix Makefile, which helps in building, packaging, and testing the Rails code. This will be used by **rake** utility supplied along with Ruby installation.

5. Examples

Subsequent chapters are based on the example taken in this chapter. In this chapter, we will create a simple but operational online library system for holding and managing the books.

This application has a basic architecture and will be built using two *ActiveRecord* models to describe the types of data that is stored in your database:

- **Books** - They describe an actual listing of the books.
- **Subject** - This is used to group books together.

Workflow for Creating Rails Applications

A recommended workflow for creating a Rails Application is as follows:

- Use the **rails** command to create the basic skeleton of the application.
- Create a database with necessary definition in the MySQL server to hold your data.
- Configure the application to know where your database is located and specify the login credentials for it.
- Create Rails Active Records (Models), because they are the business objects you'll be working with in your controllers.
- Generate Migrations that simplify the creating and maintaining of database tables and columns.
- Write Controller Code to put a life in your application.
- Create Views to present your data through User Interface.

So, let us start with creating our library application.

Creating an Empty Rails Application

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

1. Go to ruby installation directory to create your application.

2. Run the following command to create a skeleton for our library application.

```
C:\ruby> rails -d mysql library
```

This will create a subdirectory for the library application containing a complete directory tree of folders and files for an empty Rails application. Check a complete directory structure of the application. Check **Rails Directory Structure** for more detail.

Here, we are using **-d mysql** option to specify our interest to use MySQL database. We can specify any other database name like *oracle* or *postgres* using **-d** option. By default, Rails uses **SQLite** database.

Most of our development work will be creating and editing files in the **~/library/app** subdirectories. Here's a quick rundown on how to use them:

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

Starting Web Server

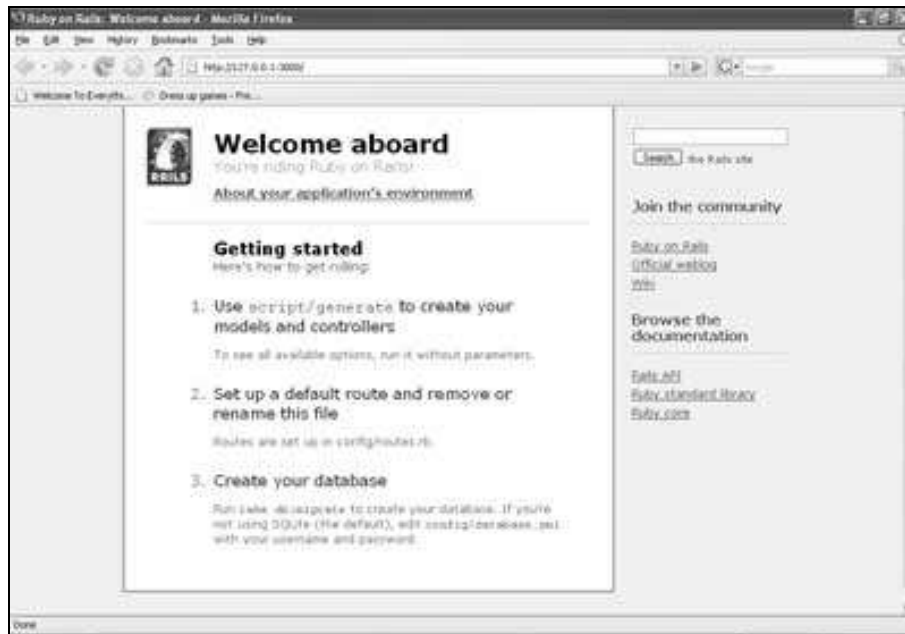
Rails web application can run virtually under any web server, but the most convenient way to develop and test a Rails web application is to use the built-in WEBrick web server. Let's start this web server and then browse to our empty library application.

This server will be started from the application directory as follows. It runs on port number 3000:

```
C:\> cd ruby\library  
C:\ruby\library> ruby script/server
```

It will start your WEBrick web server listening for Web Requests at port number 3000 at local machine.

Now open your browser and browse to **http://127.0.0.1:3000**. If everything goes fine, then you should see a greeting message from WEBrick. Following is the screen for a successful setup:



If you do not get a greeting message as above, it means there is something wrong with your setup and you need to fix it before proceeding ahead.

What is Next?

The next chapter explains how to create databases for your application and what is the configuration required to access these created databases.

Further, we will see what is Rail Migration and how it is used to maintain database tables.

6. Database Setup

Before starting with this chapter, make sure your database server is setup and running. Ruby on Rails recommends to create three databases: a database each for development, testing, and production environment. According to convention, their names should be as follows:

- library_development
- library_production
- library_test

You should initialize all three of them and create a username and password for them with full read and write privileges. We are using **root** user ID for our application. In MySQL, a console session looks as follows:

```
mysql> create database library_development;
Query OK, 1 row affected (0.01 sec)

mysql> grant all privileges on library_development.*
to 'root'@'localhost' identified by 'password';
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

You can do the same thing for the other two databases, **library_production** and **library_test**.

Configuring database.yml

At this point, you need to let Rails know about the username and password for the databases. You do this in the file **database.yml**, available in the **C:\ruby\library\config** subdirectory of Rails Application you created. This file has live configuration sections for MySQL databases. In each of the sections you use, you need to change the username and password lines to reflect the permissions on the databases you've created.

When you finish, it should look something like:

```
development:
  adapter: mysql
  database: library_development
  username: root
  password: [password]
  host: localhost
test:
  adapter: mysql
  database: library_test
  username: root
  password: [password]
  host: localhost
production:
  adapter: mysql
  database: library_production
  username: root
  password: [password]
  host: localhost
```

NOTE: You can use similar setting for other databases if you are using any other database except MySQL.

What is Next?

The next two chapters explain how to model your database tables and how to manage those using Rails Migrations.

7. Active Records

Rails Active Record is the Object/Relational Mapping (ORM) layer supplied with Rails. It closely follows the standard ORM model, which is as follows:

- tables map to classes,
- rows map to objects, and
- columns map to object attributes.

Rails Active Records provides an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records. Ruby method names are automatically generated from the field names of database tables.

Each Active Record object has CRUD (**C**reate, **R**ead, **U**ppdate, and **D**eleate) methods for database access. This strategy allows simple designs and straightforward mappings between database tables and application objects.

Translating a Domain Model into SQL

Translating a domain model into SQL is generally straightforward, as long as you remember that you have to write Rails-friendly SQL. In practical terms, you have to follow certain rules such as:

- Each entity (such as book) gets a table in the database named after it, but in the plural (books).
- Each such entity-matching table has a field called id, which contains a unique integer for each record inserted into the table.
- Given entity x and entity y, if entity y belongs to entity x, then table y has a field called x_id.
- The bulk of the fields in any table, store the values for that entity's simple properties (anything that's a number or a string).

Creating Active Record Files

To create the Active Record files for our entities for library application, introduced in the previous chapter, issue the following command from the top level of the application directory.

```
C:\ruby\library\> ruby script/generate model Book
```

```
C:\ruby\library\> ruby script/generate model Subject
```

You're telling the generator to create models called Book and Subject to store instances of books and subjects. Notice that you are capitalizing Book and Subject and using the singular form. This is a Rails paradigm that you should follow each time you create a model.

When you use the generate tool, Rails creates the actual model file that holds all the methods unique to the model and the business rules you define, a unit test file for performing test-driven development, a sample data file (called fixtures) to use with the unit tests, and a Rails migration that makes creating database tables and columns easy.

Apart from creating many other files and directories, this will create files named **book.rb** and **subject.rb** containing a skeleton definition in **app/models** directory.

Content available in book.rb:

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

Content available in subject.rb:

```
class Subject < ActiveRecord::Base
end
```

Creating Associations between Models

When you have more than one model in your Rails application, you would need to create a connection between those models. You can do this via associations. Active Record supports three types of associations:

- **one-to-one:** A one-to-one relationship exists when one item has exactly one of another item. For example, a person has exactly one birthday or a dog has exactly one owner.
- **one-to-many:** A one-to-many relationship exists when a single object can be a member of many other objects. For instance, one subject can have many books.
- **many-to-many:** A many-to-many relationship exists when the first object is related to one or more of a second object, and the second object is related to one or many of the first object.

You indicate these associations by adding declarations to your models: `has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`.

Now you need to tell Rails what relationships you want to establish within the library data system. To do so, modify `book.rb` and `subject.rb` to look like this:

```
class Book < ActiveRecord::Base
  belongs_to :subject
end
```

We have used a singular subject in the above example, because one Book can belong to a single Subject.

```
class Subject < ActiveRecord::Base
  has_many :books
end
```

We have used plural books in the above example, because one subject can have multiple books.

Implementing Validations

The implementation of validations is done in a Rails model. The data you are entering into the database is defined in the actual Rails model, so it only makes sense to define what valid data entails in the same location.

Open **book.rb** and put the following validations:

```
class Book < ActiveRecord::Base
  belongs_to :subject
  validates_presence_of :title
  validates_numericality_of :price, :message=>"Error Message"
end
```

- **validates_presence_of** - Protects "NOT NULL" fields against missing user input.
- **validates_numericality_of** - Prevents the user from entering non-numeric data.

Besides the validations mentioned above, there are some other common validations. Check **Rails Quick Guide**.

What is Next?

In the next chapter, we will learn Rails Migration, which allows you to use Ruby to define changes to your database schema, making it possible to use a version control system to keep things synchronized with the actual code.

8. Migrations

Rails Migration uses Ruby to define changes to your database schema, making it possible to use a version control system to keep things synchronized with the actual code.

It has many uses, such as:

- Teams of developers - if one person makes a schema change, the other developers just need to update, and run "rake migrate".
- Production servers - run "rake migrate" when you roll out a new release to bring the database up to date as well.
- Multiple machines - if you develop on both a desktop and a laptop, or in more than one location, migrations can help you keep them all synchronized.

What can Rails Migration Do?

- `create_table(name, options)`
- `drop_table(name)`
- `rename_table(old_name, new_name)`
- `add_column(table_name, column_name, type, options)`
- `rename_column(table_name, column_name, new_column_name)`
- `change_column(table_name, column_name, type, options)`
- `remove_column(table_name, column_name)`
- `add_index(table_name, column_name, index_type)`
- `remove_index(table_name, column_name)`

Migrations support all the basic data types: string, text, integer, float, date-time, timestamp, time, date, binary and Boolean:

- **string** - is for small data types such as a title.
- **text** - is for longer pieces of textual data, such as the description.
- **integer** - is for whole numbers.
- **float** - is for decimals.
- **date-time and timestamp** - stores the date and time into a column.

- **date and time** - stores either the date only or time only.
- **binary** - is for storing data such as images, audio, or movies.
- **boolean** - is for storing true or false values.

Valid column options are:

- **limit** (:limit => "50")
- **default** (:default => "blah")
- **null** (:null => false implies NOT NULL)

NOTE: The activities done by Rails Migration can be done using any front-end GUI or direct on SQL prompt, but Rails Migration makes all those activities very easy.

See the Rails API for details on these.

Create the Migrations

Here is the generic syntax for creating a migration:

```
C:\ruby\application> ruby script/generate migration table_name
```

This will create the file db/migrate/001_table_name.rb. A migration file contains basic Ruby syntax that describes the data structure of a database table.

NOTE: Before running migration generator, it is recommended to clean the existing migrations generated by model generators.

We will create two migrations corresponding to our three tables - **books and subjects**.

```
C:\ruby> cd library
C:\ruby\library> ruby script/generate migration books
C:\ruby\library> ruby script/generate migration subjects
```

Notice that you are using lowercase for book and subject and using the plural form while creating migrations. This is a Rails paradigm that you should follow each time you create a Migration.

Edit the Code to Tell it What to Do

Go to db/migrate subdirectory of your application and edit each file one by one using any simple text editor.

Modify 001_books.rb as follows:

The ID column will be created automatically, so don't do it here as well.

```
class Books < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.column :title, :string, :limit => 32, :null => false
      t.column :price, :float
      t.column :subject_id, :integer
      t.column :description, :text
      t.column :created_at, :timestamp
    end
  end

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

The method **self.up** is used when migrating to a new version, **self.down** is used to roll back any changes if needed. At this moment, the above script will be used to create the **books** table.

Modify 002_subjects.rb as follows:

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

    Subject.create :name => "Physics"
    Subject.create :name => "Mathematics"
    Subject.create :name => "Chemistry"
    Subject.create :name => "Psychology"
    Subject.create :name => "Geography"
  end
end
```

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

The above script will be used to create **subjects** table; it will create five records in the subjects table.

Run the Migration

Now that you have created all the required migration files, it is time to execute them against the database. To do this, go to the command prompt and open the library directory in which the application is located, and then type **rake migrate** as follows:

```
C:\ruby\library> rake db:migrate
```

This will create a "schema_info" table if it doesn't exist, which tracks the current version of the database. Each new migration will be a new version, and any new migrations will be run, until your database is at the current version.

Rake is a Ruby build program similar to Unix *make* program that Rails takes advantage of, to simplify the execution of complex tasks such as updating a database's structure etc.

Running Migrations for Production and Test Databases

If you would like to specify what rails environment to use for the migration, use the RAILS_ENV shell variable.

For example:

```
C:\ruby\library> set RAILS_ENV=production
C:\ruby\library> rake db:migrate
C:\ruby\library> set RAILS_ENV=test
C:\ruby\library> rake db:migrate
C:\ruby\library> set RAILS_ENV=development
C:\ruby\library> rake db:migrate
```

NOTE: On Unix, use "export RAILS_ENV=production" instead of the set command.

What is Next?

Now we have our database and the required tables available. In the two subsequent chapters, we will explore two important components called Controller (ActionController) and View (ActionView).

1. Creating Controllers (Action Controller)
2. Creating Views (Action View)

9. Controllers

The Rails controller is the logical center of your application. It coordinates the interaction between the user, the views, and the model. The controller is also a home to a number of important ancillary services.

- It is responsible for routing external requests to internal actions. It handles people-friendly URLs extremely well.
- It manages caching, which can give applications orders-of-magnitude performance boosts.
- It manages helper modules, which extend the capabilities of the view templates without bulking up their code.
- It manages sessions, giving users the impression of an ongoing interaction with our applications.

The process for creating a controller is very easy, and it's similar to the process we've already used for creating a model. We will create just one controller here:

```
C:\ruby\library\> ruby script/generate controller Book
```

Notice that you are capitalizing Book and using the singular form. This is a Rails paradigm that you should follow each time you create a controller.

This command accomplishes several tasks, of which the following are relevant here:

- It creates a file called `app/controllers/book_controller.rb`.

If you will have a look at `book_controller.rb`, you will find it as follows:

```
class BookController < ApplicationController
end
```

Controller classes inherit from *ApplicationController*, which is the other file in the controllers folder: **application.rb**.

The *ApplicationController* contains code that can be run in all your controllers and it inherits from Rails *ActionController::Base* class.

You don't need to worry with the *ApplicationController* as of yet, so let us just define a few method stubs in **book_controller.rb**. Based on your requirement, you could define any number of functions in this file.

Modify the file to look like the following and save your changes. Note that it is up to you what name you want to give to these methods, but better to give relevant names.

```
class BookController < ApplicationController
  def list
  end
  def show
  end
  def new
  end
  def create
  end
  def edit
  end
  def update
  end
  def delete
  end
end
```

Now let's implement all the methods one by one.

Implementing the list Method

The list method gives you a printout of all the books in the database. This functionality will be achieved by the following lines of code.

```
def list
  @books = Book.find(:all)
end
```

The `@books = Book.find(:all)` line in the list method tells Rails to search the books table and store each row it finds in the @books instance object.

Implementing the show Method

The show method displays only further details on a single book. This functionality will be achieved by the following lines of code.


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

The show method's `@books = Book.find(params[:id])` line tells Rails to find only the book that has the id defined in `params[:id]`.

The *params* object is a container that enables you to pass values between the method calls. For example, when you're on the page called by the list method, you can click a link for a specific book, and it passes the id of that book via the params object so that show can find the specific book.

Implementing the new Method

The new method lets Rails know that you will create a new object. Just add the following code in this method.

```
def new
  @book = Book.new
  @subjects = Subject.find(:all)
end
```

The above method will be called when you will display a page to the user to take user input. Here the second line grabs all the subjects from the database and puts them in an array called `@subjects`.

Implementing the create Method

Once you take the user input using HTML form, it is time to create a record into the database. To achieve this, edit the create method in the `book_controller.rb` to match the following:

```
def create
  @book = Book.new(params[:book])
  if @book.save
    redirect_to :action => 'list'
  else
    @subjects = Subject.find(:all)
    render :action => 'new'
  end
end
```

```
end
```

The first line creates a new instance variable called `@book` that holds a `Book` object built from the data the user submitted. The data was passed from the new method to create using the `params` object.

The next line is a conditional statement that redirects the user to the **list** method if the object saves correctly to the database. If it doesn't save, the user is sent back to the new method. The `redirect_to` method is similar to performing a meta refresh on a web page and it automatically forwards you to your destination without any user interaction.

Then `@subjects = Subject.find(:all)` is required in case it does not save data successfully and it becomes similar case as with new option.

Implementing the edit Method

The edit method looks nearly identical to the show method. Both methods are used to retrieve a single object based on its id and display it on a page. The only difference is that the show method is not editable.

```
def edit
  @book = Book.find(params[:id])
  @subjects = Subject.find(:all)
end
```

This method will be called to display data on the screen to be modified by the user. The second line grabs all the subjects from the database and puts them in an array called `@subjects`.

Implementing the update Method

This method will be called after the edit method when user modifies a data and wants to update the changes into the database. The update method is similar to the create method and will be used to update existing books in the database.

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to :action => 'show', :id => @book
  else
    @subjects = Subject.find(:all)
    render :action => 'edit'
  end
end
```

```

    end
  end
end

```

The `update_attributes` method is similar to the `save` method used by `create`, but instead of creating a new row in the database, it overwrites the attributes of the existing row.

Then `@subjects = Subject.find(:all)` line is required in case it does not save data successfully, then it becomes similar to the edit option.

Implementing the delete Method

If you want to delete a record in to the database then you will use this method. Implement this method as follows.

```

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

```

The first line finds the classified based on the parameter passed via the `params` object and then deletes it using the `destroy` method. The second line redirects the user to the list method using a `redirect_to` call.

Additional Methods to Display Subjects

Assume you want to give a facility to your users to browse all the books based on a given subject. You can create a method inside `book_controller.rb` to display all the subjects. Assume method name is **show_subjects**:

```

def show_subjects
  @subject = Subject.find(params[:id])
end

```

Finally, your **book_controller.rb** file will look like as follows:

```

class BookController < ApplicationController
  def list
    @books = Book.find(:all)
  end
  def show
    @book = Book.find(params[:id])
  end
end

```

```
end
def new
  @book = Book.new
  @subjects = Subject.find(:all)
end
def create
  @book = Book.new(params[:book])
  if @book.save
    redirect_to :action => 'list'
  else
    @subjects = Subject.find(:all)
    render :action => 'new'
  end
end
def edit
  @book = Book.find(params[:id])
  @subjects = Subject.find(:all)
end
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to :action => 'show', :id => @book
  else
    @subjects = Subject.find(:all)
    render :action => 'edit'
  end
end
def delete
  Book.find(params[:id]).destroy
  redirect_to :action => 'list'
end
def show_subjects
  @subject = Subject.find(params[:id])
```

```
end  
end
```

Now, save your controller file and come out for the next assignment.

What is Next?

You have created almost all the methods, which will work on backend. Next, we will create a code to generate screens to display data and to take input from the user.

10. Views

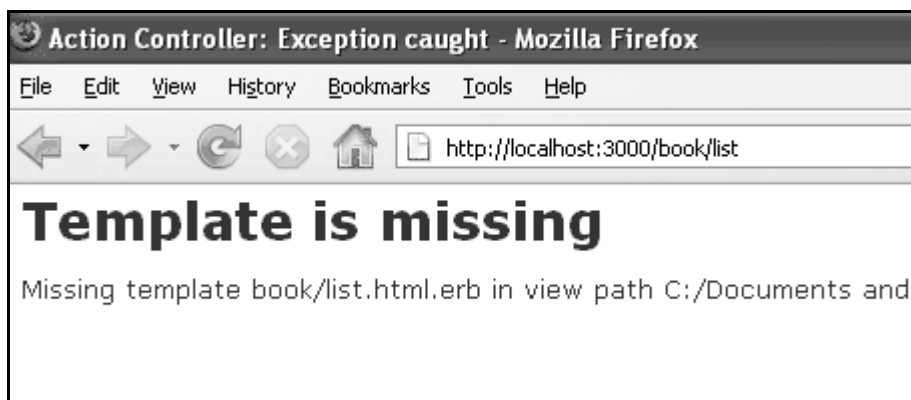
A Rails View is an ERb program that shares data with controllers through mutually accessible variables.

If you look in the app/views directory of the library application, you will see one subdirectory for each of the controllers we have created: book. Each of these subdirectories was created automatically when the same-named controller was created with the generate script.

Now, assuming your web server is up and running, provide the following input in your browser's address box:

```
http://localhost:3000/book/list
```

You get the following error message because you have not defined any view file for any method defined in the controller.



Rails lets you know that you need to create the view file for the new method. Each method you define in the controller needs to have a corresponding RHTML file, with the same name as the method, to display the data that the method is collecting.

So let us create view files for all the methods we have defined in book_controller.rb.

Creating View File for list Method

Create a file called list.rhtml using your favorite text editor and save it to app/views/book. After creating and saving the file, refresh your web browser. You should see a blank page; if you don't, check the spelling of your file and make sure that it is exactly the same as your controller's method.

Now, to display the actual content, let us put the following code into list.rhtml.

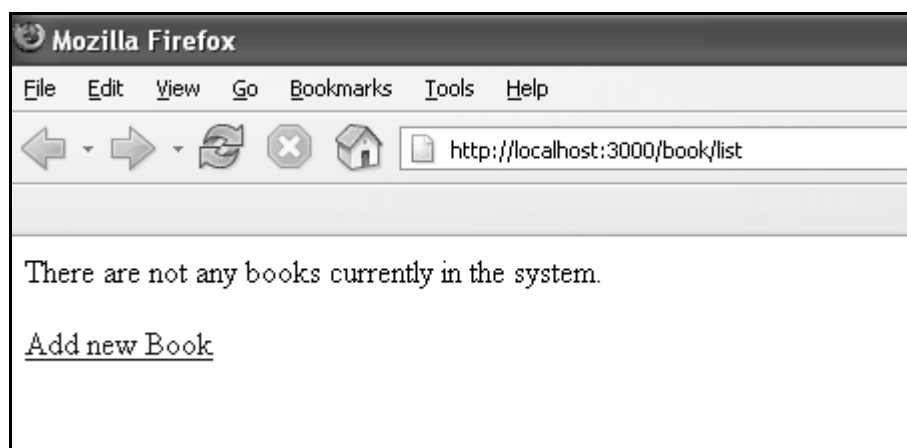
38

```
<% if @books.blank? %>
<p>There are not any books currently in the system.</p>
<% else %>
<p>These are the current books in our system</p>
<ul id="books">
<% @books.each do |c| %>
<li><%= link_to c.title, {:action => 'show', :id => c.id} -%></li>
<% end %>
</ul>
<% end %>
<p><%= link_to "Add new Book", {:action => 'new' }%></p>
```

The code to be executed is to check whether the `@books` array has any objects in it. The **.blank?** method returns true if the array is empty, and false if it contains any objects. This `@books` object was created in controller inside the list method.

The code between the `<%= %>` tags is a **link_to** method call. The first parameter of `link_to` is the text to be displayed between the `<a>` tags. The second parameter is what action is called, when the link is clicked. In this case, it is the show method. The final parameter is the id of the book that is passed via the params object.

Now, try refreshing your browser and you should get the following screen because we don't have any book in our library.



Creating View File for new Method

Till now, we don't have any book in our library. We have to create a few books in the system. So, let us design a view corresponding to the **new** method defined in `book_controller.rb`.

Create a file called `new.rhtml` using your favorite text editor and save it to `app/views/book`. Add the following code to the `new.rhtml` file.

```
<h1>Add new book</h1>
<%= start_form_tag :action => 'create' %>
<p><label for="book_title">Title</label>:
<%= text_field 'book', 'title' %></p>
<p><label for="book_price">Price</label>:
<%= text_field 'book', 'price' %></p>
<p><label for="book_subject">Subject</label>:
<%= collection_select(:book,:subject_id,@subjects,:id,:name) %></p>
<p><label for="book_description">Description</label><br/>
<%= text_area 'book', 'description' %></p>
<%= submit_tag "Create" %>
<%= end_form_tag %>
<%= link_to 'Back', {:action => 'list'} %>
```

Here the **start_form_tag()** method interprets the Ruby code into a regular HTML `<form>` tag using all the information supplied to it. This tag, for example, outputs the following HTML:

```
<form action="/book/create" method="post">
```

The next method is **text_field** that outputs an `<input>` text field. The parameters for `text_field` are object and field name. In this case, the object is `book` and the name is `title`.

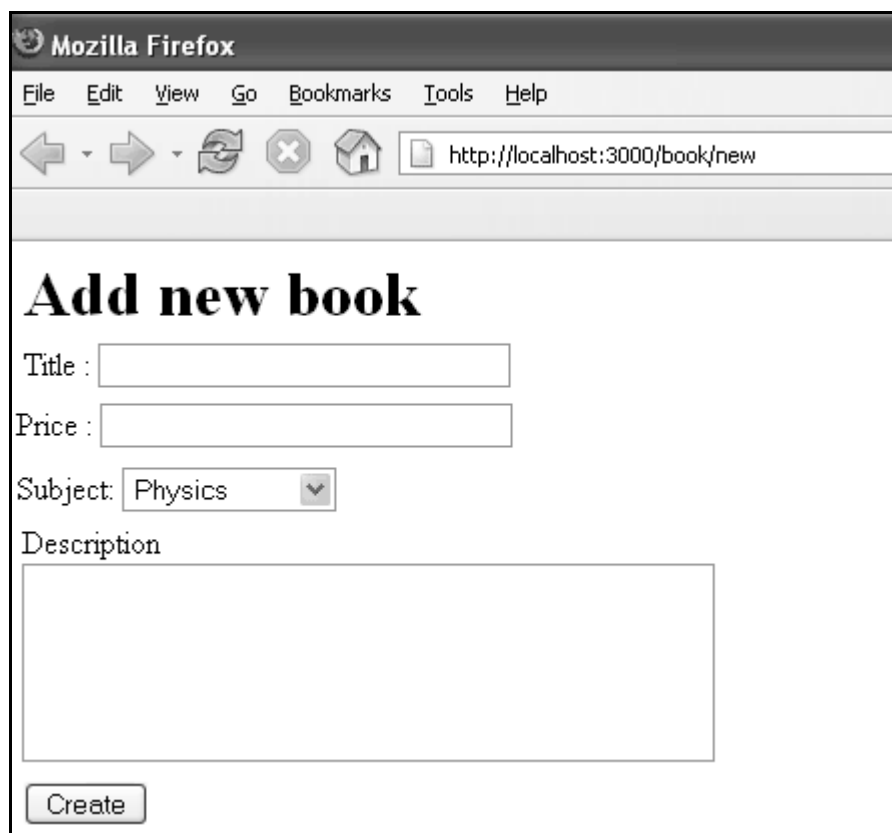
The Rails method called **collection_select** creates an HTML select menu built from an array, such as the `@books` one. There are five parameters, which are as follows:

- **:book** - The object you are manipulating. In this case, it's a book object.
- **:subject_id** - The field that is populated when the book is saved.
- **@books** - The array you are working with.

- **:id** - The value that is stored in the database. In terms of HTML, this is the `<option>` tag's value parameter.
- **:name**- The output that the user sees in the pull-down menu. This is the value between the `<option>` tags.

The next used is **submit_tag**, which outputs an `<input>` button that submits the form. Finally, there is the **end_form_tag** method that simply translates into `</form>`.

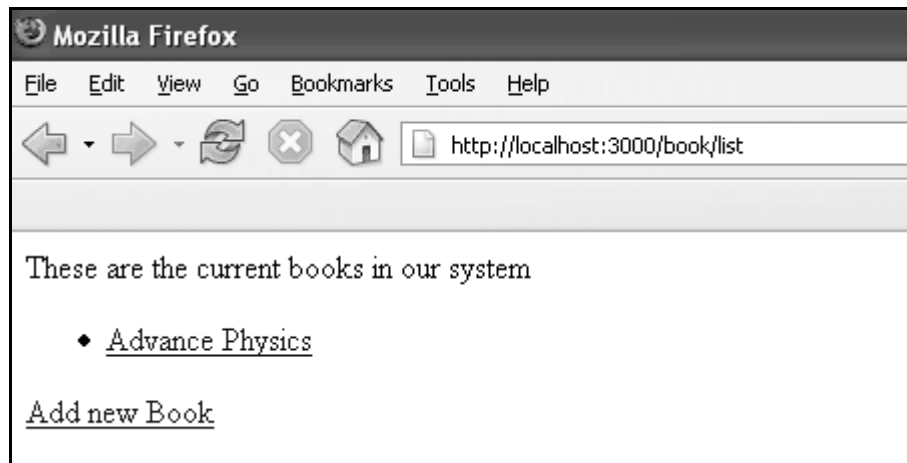
Go to your browser and visit <http://localhost:3000/book/new>. This will give you the following screen.



The screenshot shows a Mozilla Firefox browser window with the address bar displaying `http://localhost:3000/book/new`. The page content is a form titled "Add new book". The form contains the following elements:

- Title :** A text input field.
- Price :** A text input field.
- Subject:** A dropdown menu with "Physics" selected.
- Description**: A large text area.
- Create**: A button at the bottom of the form.

Enter some data in this form and then click the Create button. This will result in a call to **create** method, which does not need any view because this method is using either the **list** or **new** methods to view the results. When you click the *Create* button, the data should submit successfully and redirect you to the list page, in which you now have a single item listed as follows:



If you click the link, you should see another error "Template is missing" since you haven't created the template file for the show method yet.

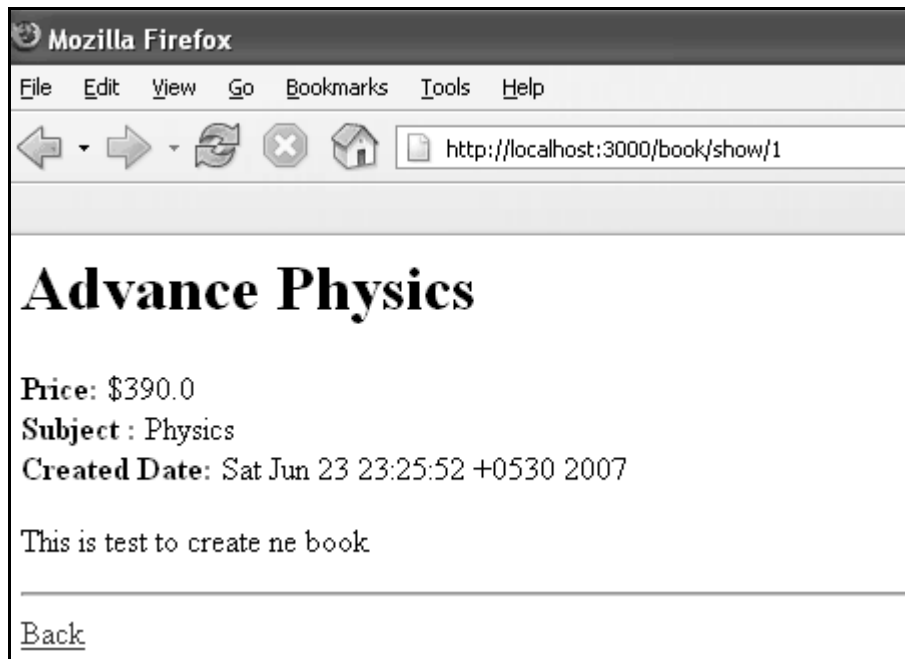
Creating View File for show Method

This method will display the complete detail about any book available in the library. Create a show.rhtml file under app/views/book and populate it with the following code:

```
<h1><%= @book.title %></h1>
<p><strong>Price: </strong> $<%= @book.price %><br />
<strong>Subject :</strong> <%= @book.subject.name %><br />
<strong>Created Date:</strong> <%= @book.created_at %><br />
</p>
<p><%= @book.description %></p>
<hr />
<%= link_to 'Back', {:action => 'list'} %>
```

This is the first time you have taken full advantage of associations, which enable you to easily pull data from related objects.

The format used is **@variable.relatedObject.column**. In this instance, you can pull the subject's name value through the @book variable using the **belongs_to** associations. If you click on any listed record, it will show you the following screen.



Creating View File for edit Method

Create a new file called edit.rhtml and save it in app/views/book. Populate it with the following code:

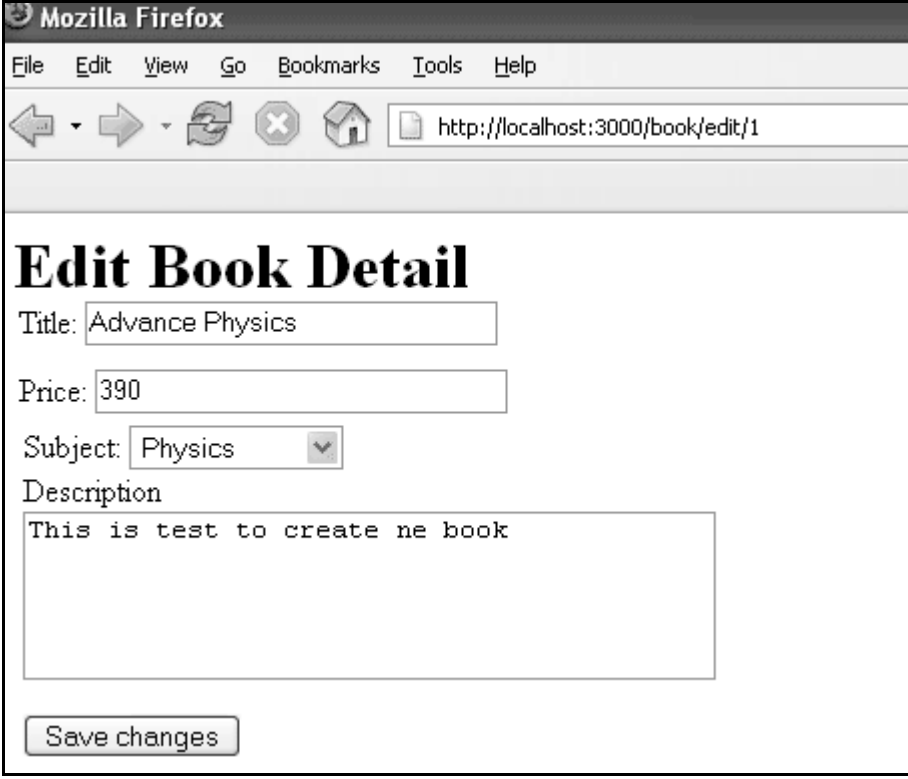
```
>
<h1>Edit Book Detail</h1>
<%= start_form_tag :action => 'update', :id => @book %>
<p><label for="book_title">Title</label>:
<%= text_field 'book', 'title' %></p>
<p><label for="book_price">Price</label>:
<%= text_field 'book', 'price' %></p>
<p><label for="book_subject">Subject</label>:
<%= collection_select(:book, :subject_id,
                      @subjects, :id, :name) %></p>
<p><label for="book_description">Description</label><br/>
<%= text_area 'book', 'description' %></p>
<%= submit_tag "Save changes" %>
<%= end_form_tag %>
<%= link_to 'Back', {:action => 'list' } %>
```

This code is very similar to the **new** method, except for the fact that action to be updated instead of creating and defining an id.

At this point, we need some modification in the **list method's** view file. Go to the `` element and modify it so that it looks as follows:

```
<li>
<%= link_to c.title, {:action => "show", :id => c.id} -%>
<b> <%= link_to 'Edit', {:action => "edit",
:id => c.id} %></b>
</li>
```

Now, try to browse books using `http://localhost:3000/book/list`. It will give you the listing of all the books along with **Edit** option. When you click the Edit option, you will have the next screen as follows:



The screenshot shows a Mozilla Firefox browser window with the address bar displaying `http://localhost:3000/book/edit/1`. The main content area is titled "Edit Book Detail" and contains a form with the following fields:

- Title:
- Price:
- Subject: (with a dropdown arrow)
- Description:

At the bottom of the form is a button labeled "Save changes".

Now, you edit this information and then click the *Save Changes* button. It will result in a call to **update** method available in the controller file and it will update all the changed attributes. Notice that the **update** method does not need any view file because it's using either **show** or **edit** methods to show its results.

Creating View File for delete Method

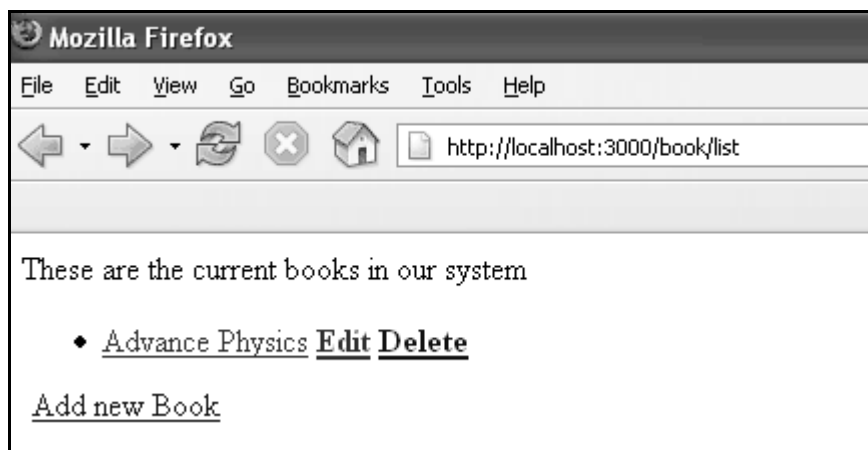
Removing information from a database using Ruby on Rails is almost too easy. You do not need to write any view code for the delete method, because this method is using **list** method to display the result. So, let's just modify list.rhtml again and add a delete link.

Go to the `` element and modify it to look like the following:

```
<li>
<%= link_to c.title, {:action => 'show', :id => c.id} -%>
<b> <%= link_to 'Edit', {:action => 'edit', :id => c.id} %></b>
<b> <%= link_to "Delete", {:action => 'delete', :id => c.id},
:confirm => "Are you sure you want to delete this item?" %></b>
</li>
```

The **:confirm** parameter presents a JavaScript confirmation box asking if you really want to perform the action. If the user clicks OK, the action proceeds, and the item is deleted.

Now, try browsing books using `http://localhost:3000/book/list`. It will give you the listing of all the books along with **Edit** and **Delete** options as follows:



Now, using the **Delete** option, you can delete any listed record.

Creating View File for show_subjects Method

Create a new file, `show_subjects.rhtml`, in the `app/views/book` directory and add the following code to it:

```
<h1><%= @subject.name -%></h1>
<ul>
```

```
<% @subject.books.each do |c| %>
<li><%= link_to c.title, :action => "show", :id => c.id -%></li>
<% end %>
</ul>
```

You are taking advantage of associations by iterating through a single subject's many books listings.

Now, modify the Subject line of show.rhtml so that the subject listing shows a link.

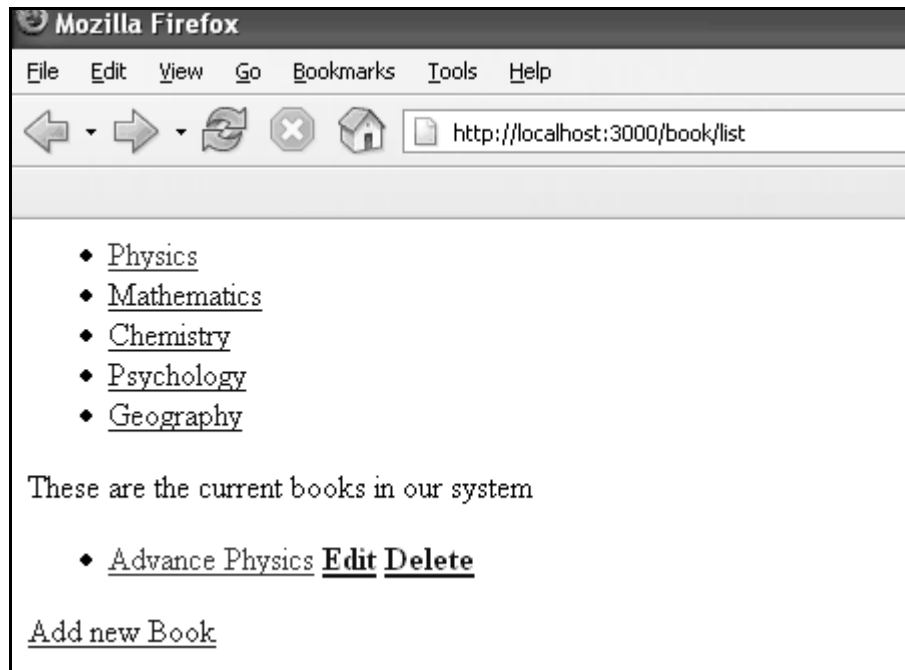
```
<strong>Subject: </strong> <%= link_to @book.subject.name,
:action => "show_subjects", :id => @book.subject.id %><br />
```

This will output a list of subject on the index page, so that users can access them directly.

Modify **list.rhtml** to add the following at the top of the file:

```
<ul id="subjects">
<% Subject.find(:all).each do |c| %>
<li><%= link_to c.name, :action => "show_subjects",
:id => c.id %></li>
<% end %>
</ul>
```

Now, try browsing books using <http://localhost:3000/book/list>. It will display all the subjects with links so that you can browse all the books related to that subject.



What is Next?

We hope you are now comfortable with all the Rails Operations.

The next chapter explains how to use **Layouts** to put your data in a better way. We will also show you how to use CSS in your Rails applications.

11. Layouts

A layout defines the surroundings of an HTML page. It's the place to define the common look and feel of your final output. Layout files reside in `app/views/layouts`.

The process involves defining a layout template and then letting the controller know that it exists and to use it. First, let's create the template.

Add a new file called `standard.rhtml` to `app/views/layouts`. You let the controllers know what template to use by the name of the file, so following a same naming scheme is advised.

Add the following code to the new `standard.rhtml` file and save your changes:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
<meta http-equiv="Content-Language" content="en-us" />
<title>Library Info System</title>
<%= stylesheet_link_tag "style" %>
</head>
<body id="library">
<div id="container">
<div id="header">
<h1>Library Info System</h1>
<h3>Library powered by Ruby on Rails</h3>
</div>
<div id="content">
<%= yield -%>
</div>
<div id="sidebar"></div>
</div>
```



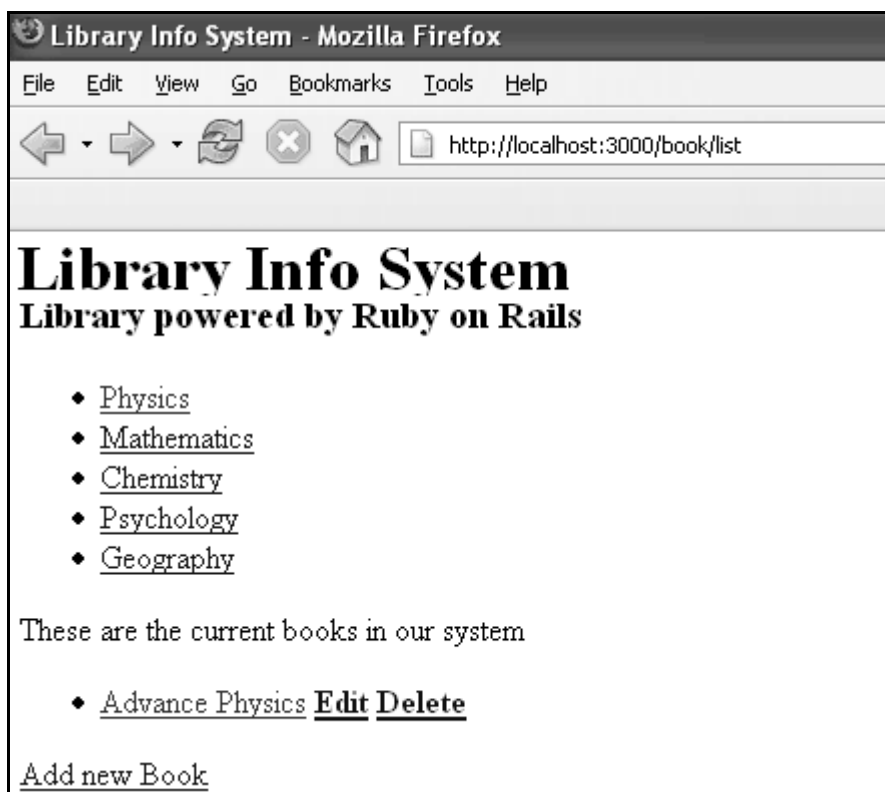
```
</body>
</html>
```

Everything you just added are standard HTML elements except the two lines with the **stylesheet_link_tag** helper method that outputs a stylesheet <link>. In this instance, we are linking the style.css stylesheet. The **yield** command lets Rails know that it should put the RHTML for the method called here.

Now open **book_controller.rb** and add the following line just below the first line:

```
class BookController < ApplicationController
  layout 'standard'
  def list
    @books = Book.find(:all)
  end
  .....
```

It directs the controller that we want to use a layout available in the standard.rhtml file. Now, try browsing books that will produce the following screen.



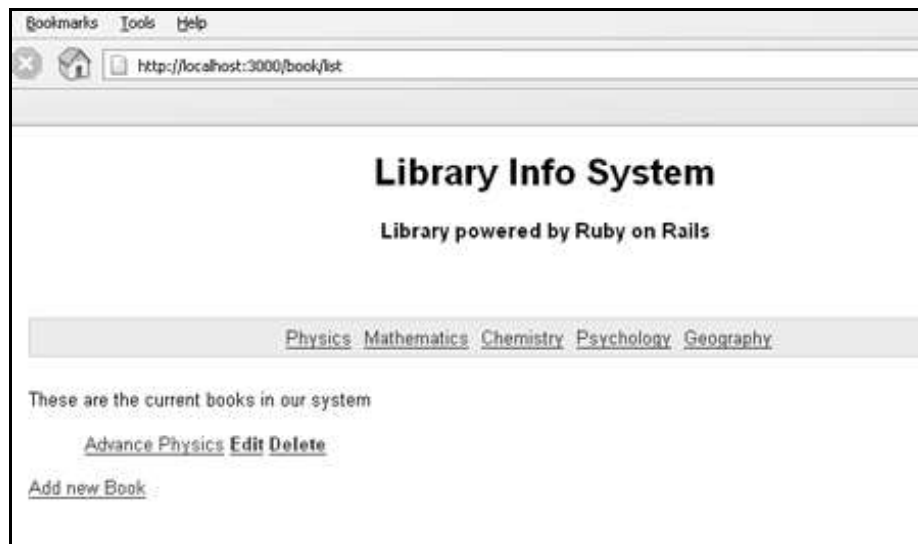
Adding Style Sheet

Till now, we have not created any style sheet, so Rails is using the default style sheet. Now, let's create a new file called `style.css` and save it in `/public/stylesheets`. Add the following code to this file.

```
body {
  font-family: Helvetica, Geneva, Arial, sans-serif;
  font-size: small;
  font-color: #000;
  background-color: #fff;
}
a:link, a:active, a:visited {
  color: #CD0000;
}
input {
  margin-bottom: 5px;
}
p {
  line-height: 150%;
}
div#container {
  width: 760px;
  margin: 0 auto;
}
div#header {
  text-align: center;
  padding-bottom: 15px;
}
div#content {
  float: left;
  width: 450px;
  padding: 10px;
}
div#content h3 {
```

```
    margin-top: 15px;
}
ul#books {
    list-style-type: none;
}
ul#books li {
    line-height: 140%;
}
div#sidebar {
    width: 200px;
    margin-left: 480px;
}
ul#subjects {
    width: 700px;
    text-align: center;
    padding: 5px;
    background-color: #ececec;
    border: 1px solid #ccc;
    margin-bottom: 20px;
}
ul#subjects li {
    display: inline;
    padding-left: 5px;
}
```

Now, refresh your browser and see the difference:



What is Next?

The next chapter explains how to develop applications with Rails Scaffolding to give user access to add, delete, and modify the records in any database.

12. Scaffolding

While you're developing Rails applications, especially those which are mainly providing you with a simple interface to data in a database, it can often be useful to use the scaffold method.

Scaffolding provides more than cheap demo thrills. Here are some benefits:

- You can quickly get code in front of your users for feedback.
- You are motivated by faster success.
- You can learn how Rails works by looking at generated code.
- You can use the scaffolding as a foundation to jumpstart your development.

Scaffolding Example

Ruby on Rails 2.0 changes the way Rails uses scaffolding. To understand *scaffolding*, let's create a database called **cookbook** and a table called **recipes**.

Creating an Empty Rails Web Application

Open a command window and navigate to where you want to create this **cookbook** web application. We used c:\ruby. Run the following command to create complete directory structure and required .yml file MySQL database.

```
C:\ruby> rails -d mysql cookbook
```

Here we are using **-d mysql** option to specify our interest to use MySQL database. We can specify any other database name like *oracle* or *postgres* using **-d** option. By default, Rails uses **SQLite** database.

Setting Up the Database

Here is the way to create database:

```
mysql> create database cookbook;
Query OK, 1 row affected (0.01 sec)

mysql> grant all privileges on cookbook.*
to 'root'@'localhost' identified by 'password';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> FLUSH PRIVILEGES;
```

```
Query OK, 0 rows affected (0.00 sec)
```

To instruct Rails to locate the database, edit the configuration file `~\cookbook\config\database.yml` and change the database name to `cookbook`. When you finish, it should look as follows:

```
development:
  adapter: mysql
  encoding: utf8
  database: cookbook
  username: root
  password: password
  host: localhost
test:
  adapter: mysql
  encoding: utf8
  database: cookbook
  username: root
  password: password
  host: localhost
production:
  adapter: mysql
  encoding: utf8
  database: cookbook
  username: root
  password: password
  host: localhost
```

NOTE: You can use similar settings for other database adapters in case you want to use any other database except MySQL.

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

Database Table Definition

Assuming the following structure for our **recipes** table:

```
id INT(11)
title VARCHAR(40)
chef VARCHAR(40)
instructions VARCHAR(255)
```

The Generated Scaffold Code

With the *scaffold* action, Rails generates all the code it needs dynamically. By running *scaffold* as a script, generate the model plus scaffolding, and the database migration script needed as well as a controller, helper, and testing support files as follows:

```
cookbook> ruby script/generate scaffold Recipe title:string \
chef:string instructions:text
```

Note a singular name *Recipe* to create a plural table name *recipes*. However, the above command will generate the following messages:

```
exists  app/models/
exists  app/controllers/
exists  app/helpers/
create  app/views/recipes
exists  app/views/layouts/
exists  test/functional/
exists  test/unit/
exists  public/stylesheets/
create  app/views/recipes/index.html.erb
create  app/views/recipes/show.html.erb
create  app/views/recipes/new.html.erb
create  app/views/recipes/edit.html.erb
create  app/views/layouts/recipes.html.erb
create  public/stylesheets/scaffold.css
create  app/controllers/recipes_controller.rb
create  test/functional/recipes_controller_test.rb
```

```

    create  app/helpers/recipes_helper.rb
    route  map.resources :recipes
dependency  model
    exists  app/models/
    exists  test/unit/
    exists  test/fixtures/
    create  app/models/recipe.rb
    create  test/unit/recipe_test.rb
    create  test/fixtures/recipes.yml
    create  db/migrate
    create  db/migrate/20080614192220_create_recipes.rb
cookbook>

```

Now, let's examine what has happened behind the scene.

The Controller

Let's look at the code behind the controller. This code is generated by the **scaffold** generator. If you open `app/controllers/recipes_controller.rb`, you will find something as follows:

```

class RecipesController < ApplicationController
  # GET /recipes
  # GET /recipes.xml
  def index
    @recipes = Recipe.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @recipes }
    end
  end

  # GET /recipes/1
  # GET /recipes/1.xml
  def show

```



```
@recipe = Recipe.find(params[:id])

respond_to do |format|
  format.html # show.html.erb
  format.xml { render :xml => @recipe }
end
end

# GET /recipes/new
# GET /recipes/new.xml
def new
  @recipe = Recipe.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @recipe }
  end
end

# GET /recipes/1/edit
def edit
  @recipe = Recipe.find(params[:id])
end

# POST /recipes
# POST /recipes.xml
def create
  @recipe = Recipe.new(params[:recipe])

  respond_to do |format|
    if @recipe.save
      flash[:notice] = 'Recipe was successfully created.'
      format.html { redirect_to(@recipe) }
    end
  end
end
```

```

        format.xml { render :xml => @recipe,
                           :status => :created, :location => @recipe }
    else
        format.html { render :action => "new" }
        format.xml { render :xml => @recipe.errors,
                           :status => :unprocessable_entity }
    end
end
end

# PUT /recipes/1
# PUT /recipes/1.xml
def update
    @recipe = Recipe.find(params[:id])

    respond_to do |format|
        if @recipe.update_attributes(params[:recipe])
            flash[:notice] = 'Recipe was successfully updated.'
            format.html { redirect_to(@recipe) }
            format.xml { head :ok }
        else
            format.html { render :action => "edit" }
            format.xml { render :xml => @recipe.errors,
                           :status => :unprocessable_entity }
        end
    end
end

# DELETE /recipes/1
# DELETE /recipes/1.xml
def destroy
    @recipe = Recipe.find(params[:id])
    @recipe.destroy
end

```

```

    respond_to do |format|
      format.html { redirect_to(recipes_url) }
      format.xml  { head :ok }
    end
  end
end
end

```

This file has all the methods implemented automatically. You can perform any Create, Read, Delete, or Edit operation using these available methods.

When a user of a Rails application selects an action, e.g., "Show" - the controller will execute any code in the appropriate section - "def show" - and then by default will render a template of the same name - "show.html.erb". This default behavior can be overwritten by overwriting the code in any template.

The controller uses ActiveRecord methods such as *find*, *find_all*, *new*, *save*, *update_attributes*, and *destroy* to move data to and from the database tables. Note that you do not have to write any SQL statements, Rails will take care of it automatically.

The Views

All the views and corresponding controller methods are created by **scaffold** command and they are available in `app/views/recipes` directory. You will have the following files in this directory:

- **index.html.erb** - This is the template file to show the default page and will be executed when you type `http://127.0.0.1:3000/recipes`.
- **new.html.erb** - This is the template to create a new recipe and will be executed whenever you will try to create a new recipe.
- **show.html.erb** - This is the template to show all the recipes in your database and will be executed whenever you will try to see all the recipes.
- **edit.html.erb** - This is the template to edit any recipe in your database and will be executed whenever you will try to edit any recipe.

We suggest you to open these files one by one and try to understand their source code.

The Migrations

You will find a migration file created in `~/cookbook/db/migrate` subdirectory. This file will have the following content:

```
class CreateRecipes < ActiveRecord::Migration
  def self.up
    create_table :recipes do |t|
      t.string :title
      t.string :chef
      t.text :instructions
      t.timestamps
    end
  end

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

To create the required file in your database, make use of helper script as follows.

```
cookbook> rake db:migrate
```

This command will create **recipes** and **schema_migrations** tables in your **cookbook** database. Before proceeding, please make sure you have the required table created successfully in your database.

Ready to Test

All the above steps bring your database table to life. It provides a simple interface to your data, and ways of:

- Creating new entries
- Editing current entries
- Viewing current entries
- Destroying current entries

When creating or editing an entry, scaffold will do all the hard work of form generation and handling. It will even provide clever form generation, supporting the following types of inputs:

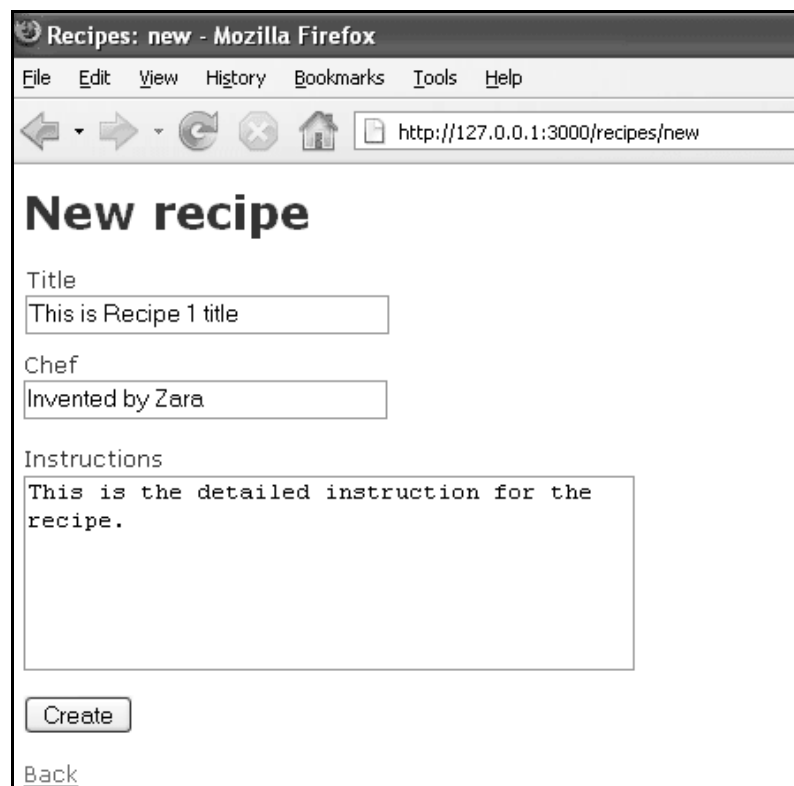
- Simple text strings
- Text areas (or large blocks of text)

- Date selectors
- Date-time selectors

Now, go to cookbook directory and run the Web Server using the following command:

```
cookbook> ruby script/server
```

Now, open a browser and navigate to <http://127.0.0.1:3000/recipes/new>. It will provide you a screen to create new entries in recipes table. A screenshot is shown below:



The screenshot shows a Mozilla Firefox browser window with the title 'Recipes: new - Mozilla Firefox'. The address bar displays 'http://127.0.0.1:3000/recipes/new'. The main content area features a form titled 'New recipe'. The form includes three input fields: 'Title' with the value 'This is Recipe 1 title', 'Chef' with the value 'Invented by Zara', and 'Instructions' with the value 'This is the detailed instruction for the recipe.'. Below the input fields is a 'Create' button. At the bottom left of the form is a 'Back' link.

Now, enter some values in the given text boxes, and press the Create button to create a new recipe. Your record is added into the recipes table and it shows the following result:



You can use either the **Edit** option to edit the recipe or the **Back** button to go to the previous page. Assuming you have pressed the **Back** button, it will display all the recipes available in your database. As we have only one record in our database, it will show you the following screen:



This screen gives you the option to see the complete details of the recipe table. In addition, it provides options to edit or even delete the table.

Enhancing the Model

Rails gives you a lot of error handling for free. To understand this, add some validation rules to the empty recipe model:

Modify `~/cookbook/app/models/recipe.rb` as follows and then test your application:

```
class Recipe < ActiveRecord::Base
  validates_length_of :title, :within => 1..20
end
```

```

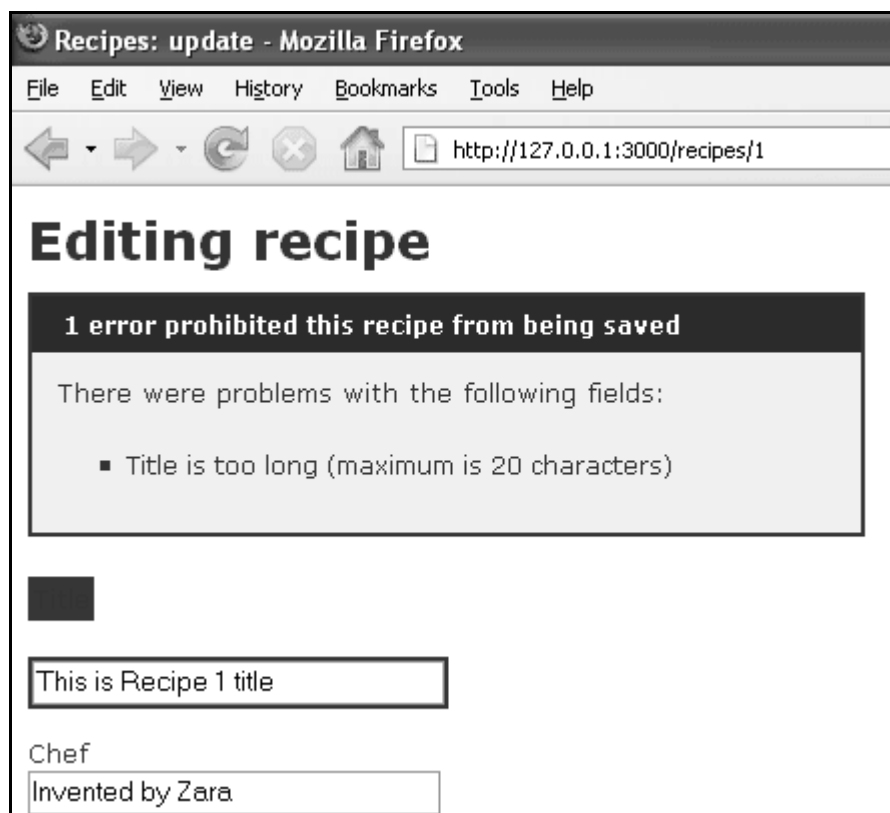
    validates_uniqueness_of :title, :message => "already exists"
  end

```

These entries will give automatic checking such as:

- **validates_length_of:** The field is not blank and not too long.
- **validates_uniqueness_of:** Duplicate values are trapped. Instead of the default Rails error message, we have given our custom message.

Here we are trying to provide a bigger title while editing the existing record. It produces the following error message, just because we have added the above validations:



How Scaffolding is Different?

If you have gone through the previous chapters, then you must have seen that we had created methods to list, show, delete, and create data, but scaffolding does that job automatically.

13. Ajax on Rails

Ajax stands for **A**synchronous **J**avaScript and **X**ML. Ajax is not a single technology; it is a suite of several technologies. Ajax incorporates the following:

- XHTML for the markup of web pages
- CSS for the styling
- Dynamic display and interaction using the DOM
- Data manipulation and interchange using XML
- Data retrieval using XMLHttpRequest
- JavaScript as the glue that meshes all this together

Ajax enables you to retrieve data for a web page without having to refresh the contents of the entire page. In the basic web architecture, the user clicks a link or submits a form. The form is submitted to the server, which then sends back a response. The response is then displayed for the user on a new page.

When you interact with an Ajax-powered web page, it loads an Ajax engine in the background. The engine is written in JavaScript and its responsibility is to both communicate with the web server and display the results to the user. When you submit data using an Ajax-powered form, the server returns an HTML fragment that contains the server's response and displays only the data that is new or changed as opposed to refreshing the entire page.

For a complete detail on AJAX you can go through our **AJAX Tutorial**.

How Rails Implements Ajax

Rails has a simple, consistent model for how it implements Ajax operations. Once the browser has rendered and displayed the initial web page, different user actions cause it to display a new web page (like any traditional web application) or trigger an Ajax operation:

- **Some trigger fires:** This trigger could be the user clicking on a button or link, the user making changes to the data on a form or in a field, or just a periodic trigger (based on a timer).
- **The web client calls the server:** A JavaScript method, *XMLHttpRequest*, sends data associated with the trigger to an action handler on the server. The data might be the ID of a checkbox, the text in an entry field, or a whole form.

- **The server does processing:** The server-side action handler (Rails controller action), does something with the data and returns an HTML fragment to the web client.
- **The client receives the response:** The client-side JavaScript, which Rails creates automatically, receives the HTML fragment and uses it to update a specified part of the current page's HTML, often the content of a <div> tag.

These steps are the simplest way to use Ajax in a Rails application, but with a little extra work, you can have the server return any kind of data in response to an Ajax request, and you can create custom JavaScript in the browser to perform more involved interactions.

AJAX Example

While discussing rest of the Rails concepts, we have taken an example of Library. There we have a table called **subject** and we had added few subjects at the time of Migration. Till now we have not provided any procedure to add and delete subjects in this table.

In this example, we will provide, list, show and create operations on subject table. If you don't have any understanding on Library Info System explained in the previous chapters, then we would suggest you to complete the previous chapters first and then continue with AJAX on Rails.

Creating Controller

Let's create a controller for subject. It will be done as follows:

```
C:\ruby\library> ruby script/generate controller Subject
```

This command creates a controller file `app/controllers/subject_controller.rb`. Open this file in any text editor and modify it to have the following content:

```
class SubjectController < ApplicationController
  layout 'standard'
  def list
  end
  def show
  end
  def create
  end
end
```

```
end
```

Now, we will discuss the implementation part of all these functions in the same way we had given in the previous chapters.

The list Method Implementation

```
def list
  @subjects = Subject.find(:all)
end
```

This is similar to the example explained earlier and will be used to list down all the subjects available in our database.

The show Method Implementation

```
def show
  @subject = Subject.find(params[:id])
end
```

This is also similar to the example explained earlier and will be used to display a particular subject corresponding to the passed ID.

The create Method Implementation

```
def create
  @subject = Subject.new(params[:subject])
  if @subject.save
    render :partial => 'subject', :object => @subject
  end
end
```

This part is a bit new. Here we are not redirecting the page to any other page; we are just rendering the changed part instead of whole page.

It happens only when using **partial**. We don't write the complete view file, instead, we will write a partial in the /app/view/subject directory. We will see it in a moment. First, let's create view files for other methods.

Creating Views

Now we will create view files for all the methods except for the create method for which we will create a partial.

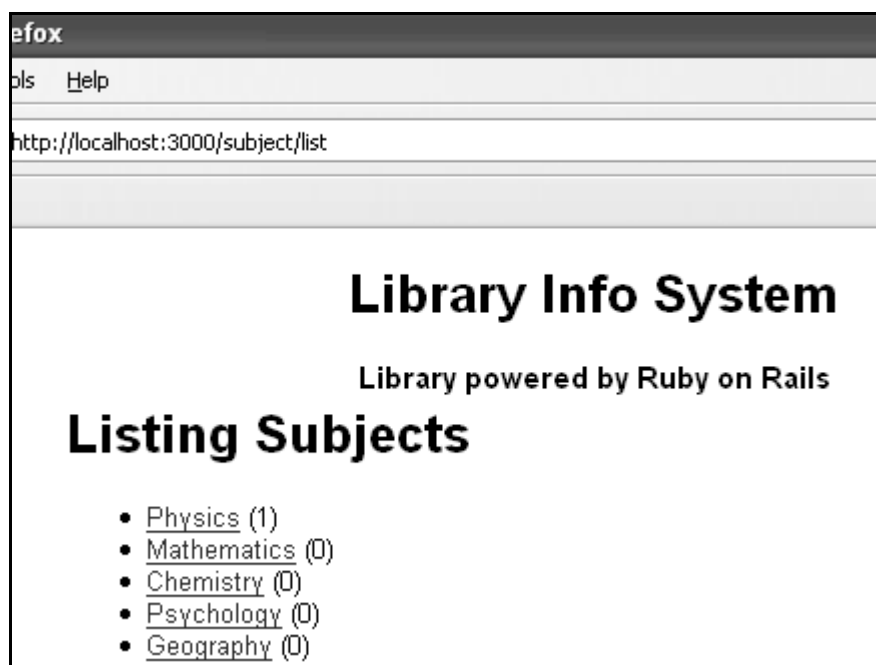
Creating View for list Method

Create a file list.rhtml in /app/view/subject and populate it with the following code.

```
<h1>Listing Subjects</h1>
<ul id="subject_list">
  <% @subjects.each do |c| %>
    <li><%= link_to c.name, :action => 'show', :id => c.id %>
      <%= "(#{c.books.count})" -%></li>
  <% end %>
</ul>
```

Here, you are iterating through the @subjects array and outputting a element containing a link to the subject it is referencing for each item in the array. Additionally, you are outputting the number of books in that specific subject inside parentheses. Rails' associations make it easy to step through a relationship and get information like this.

Now, try browsing your Subject list using <http://localhost:3000/subject/list>. It will show you the following screen.



Creating View for show Method

Create a file show.rhtml in /app/view/subject and populate it with the following code.

```
<h1><%= @subject.name -%></h1>
<ul>
<% @subject.books.each do |c| %>
<%= link_to c.title, :controller => "book",
               :action => "show",:id => c.id -%>

<% end %>
</ul>
```

Now, try clicking on any subject and you will find a listing of all the books available under that subject.

Creating View for create Method

We would not create view for the **create** method because we are using **partial** instead of view. In the next section, we will create a partial for the create method.

Adding Ajax Support

To get Ajax support in the Rails application, you need to include the necessary JavaScript files in the layout. Rails is bundled with several libraries that make using Ajax very easy. Two libraries - **prototype** and **script.aculo.us** are very popular.

To add Prototype and script.aculo.us support to the application, open up the standard.rhtml layout file in app/views/layouts, add the following line just before the </head> tag, and save your changes:

```
<%= javascript_include_tag :defaults %>
```

This includes both the Prototype and script.aculo.us libraries in the template, so their effects will be accessible from any of the views.

Now, add the following code at the bottom of app/views/subject/list.rhtml.

```
<p id="add_link"><%= link_to_function("Add a Subject",
"Element.remove('add_link'); Element.show('add_subject')")%></p>

<div id="add_subject" style="display:none;">
```

```

<%= form_remote_tag(:url => {:action => 'create'},
  :update => "subject_list", :position => :bottom,
  :html => {:id => 'subject_form'})%>
Name: <%= text_field "subject", "name" %>
<%= submit_tag 'Add' %>
<%= end_form_tag %>
</div>

```

We are using `link_to_function` instead of `link_to` method because the `link_to_function` method enables you to harness the power of the Prototype JavaScript library to do some neat DOM manipulations.

The second section is the creation of the `add_subject` `<div>`. Notice that you set its visibility to be hidden by default using the CSS display property. The preceding `link_to_function` is what will change this property and show the `<div>` to the user to take input required to add a new subject.

Next, you are creating the Ajax form using the **`form_remote_tag`**. This Rails helper is similar to the **`start_form_tag`** tag, but it is used here to let the Rails framework know that it needs to trigger an Ajax action for this method. The `form_remote_tag` takes the **`:action`** parameter just like `start_form_tag`.

You also have two additional parameters: **`:update`** and **`:position`**.

- The **`:update`** parameter tells Rails' Ajax engine which element to update based on its id. In this case, it's the `` tag.
- The **`:position`** parameter tells the engine where to place the newly added object in the DOM. You can set it to be at the bottom of the unordered list (`:bottom`) or at the top (`:top`).

Next, you create the standard form fields and submit buttons as before and then wrap things up with an `end_form_tag` to close the `<form>` tag. Make sure that things are semantically correct and valid XHTML.

Creating Partial for create Method

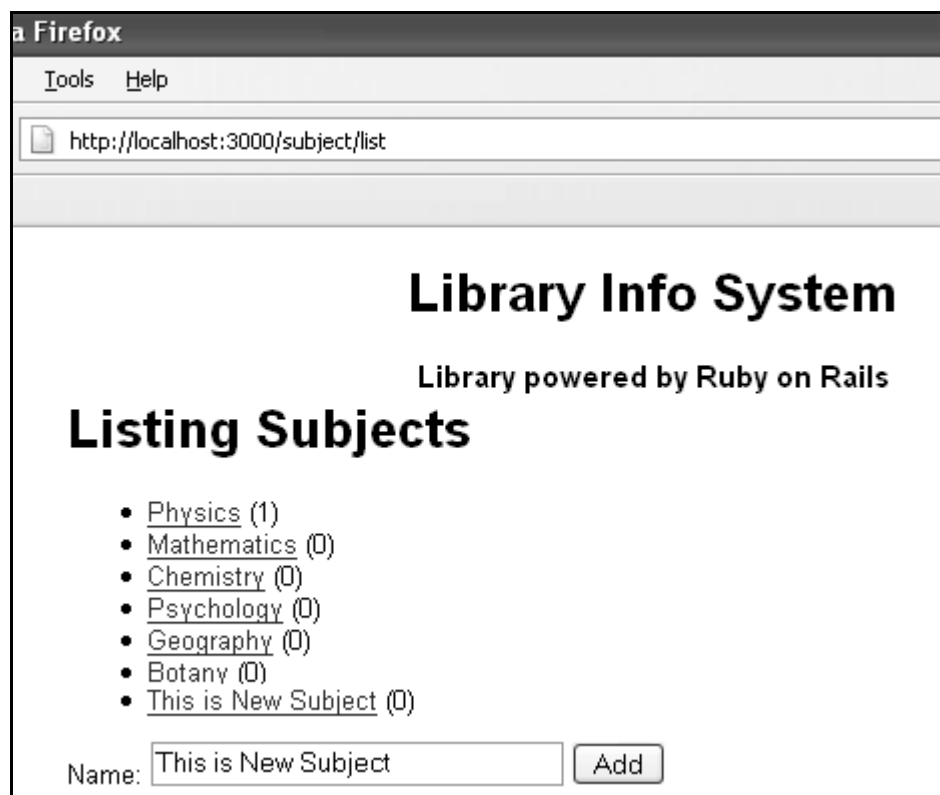
We are calling the `create` method while adding a subject, and inside this create method, we are using one partial. Let's implement this **partial** before going for actual practical.

Under `app/views/subject`, create a new file called **`_subject.rhtml`**. Notice that all the partials are named with an underscore (`_`) at the beginning.

Add the following code into this file:

```
<li id="subject_<%= subject.id %>">
  <%= link_to subject.name, :action => 'show', :id => subject.id %>
  <%= "(#{subject.books.count})" -%>
</li>
```

You are done now and can easily add several subjects without having to wait for the page to refresh after each subject is added. Now, try browsing your Subject list using <http://localhost:3000/subject/list>. It will show you the following screen. Try to add some subject.



When you press the Add button, subject would be added at the bottom of all the available subjects and you would not have a feel of page refresh.

14. File Uploading

You may have a requirement in which you want your site visitors to upload a file on your server. Rails makes it very easy to handle this requirement. Now, we will proceed with a simple and small Rails project.

As usual, let's start off with a new Rails application called **upload**. Let's create a basic structure of the application by using simple rails command.

```
C:\ruby> rails upload
```

Let's decide where you would like to save your uploaded files. Assume this is **data** directory inside your public section. So, create this directory and check the permissions.

```
C:\ruby> cd upload
C:\ruby> mkdir upload\public\data
```

Our next step will be as usual, to create controller and models.

Creating the Model

As this is not a database-based application, we can keep whatever name is comfortable to us. Assume we have to create a **DataFile** model.

```
C:\ruby> ruby script/generate model DataFile
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/data_file.rb
create  test/unit/data_file_test.rb
create  test/fixtures/data_files.yml
create  db/migrate
create  db/migrate/001_create_data_files.rb
```

Now, we will create a method called **save** in **data_file.rb** model file. This method will be called by the application controller.

```
class DataFile < ActiveRecord::Base
  def self.save(upload)
```

```

    name = upload['datafile'].original_filename
    directory = "public/data"
    # create the file path
    path = File.join(directory, name)
    # write the file
    File.open(path, "wb") { |f| f.write(upload['datafile'].read) }
  end
end

```

The above function will take the CGI object **upload** and will extract the uploaded file name using the helper function **original_filename** and finally, it will store the uploaded file into the "public/data" directory. You can call the helper function **content_type** to know the media type of the uploaded file.

Here **File** is a ruby object and **join** is a helper function that will concatenate the directory name along with the file name and will return the full file path.

Next, to open a file in write mode, we are using the open helper function provided by the **File** object. Further, we are reading data from the passed data file and writing into output file.

Creating Controller

Now, let's create a controller for our upload project:

```

C:\ruby> ruby script/generate controller Upload

exists  app/controllers/
exists  app/helpers/
create  app/views/upload
exists  test/functional/
create  app/controllers/upload_controller.rb
create  test/functional/upload_controller_test.rb
create  app/helpers/upload_helper.rb

```

Now, we will create two controller functions. The first function **index** will call a view file to take user input, and the second function **uploadFile** takes file information from the user and passes it to the 'DataFile' model. We set the upload directory to the 'uploads' directory we created earlier "directory = 'data'".


```

class UploadController < ApplicationController
  def index
    render :file => 'app\views\upload\uploadfile.rhtml'
  end
  def uploadFile
    post = DataFile.save(params[:upload])
    render :text => "File has been uploaded successfully"
  end
end

```

Here, we are calling the function defined in the model file. The **render** function is being used to redirect to view file as well as to display a message.

Creating View

Finally, we will create a view file **uploadfile.rhtml**, which we have mentioned in the controller. Populate this file with the following code:

```

<h1>File Upload</h1>
<%= start_form_tag ({:action => 'uploadFile'},
                    :multipart => true) %>
<p><label for="upload_file">Select File</label> :
<%= file_field 'upload', 'datafile' %></p>
<%= submit_tag "Upload" %>
<%= end_form_tag %>

```

Here everything is same what we have explained in the earlier chapters. The only new tag is **file_field**, which will create a button to select a file from user's computer.

By setting the multipart parameter to true, you ensure that your action properly passes along the binary data from the file.

Here, an important point to note is that we have assigned "**uploadFile**" as the method name in **:action**, which will be called when you click the **Upload** button.

It will show you a screen as follows:



Now, you select a file and upload it. This file will be uploaded into `app/public/data` directory with the actual file name and a message will be displayed saying that "File has been uploaded successfully".

NOTE: If a file with the same name already exists in your output directory, then it will be overwritten.

Files Uploaded from Internet Explorer

Internet Explorer includes the entire path of a file in the filename sent, so the **original_filename** routine will return something like:

```
C:\Documents and Files\user_name\Pictures\My File.jpg
```

Instead of just:

```
My File.jpg
```

This is easily handled by **File.basename**, which strips out everything before the filename.

```
def sanitize_filename(file_name)
  # get only the filename, not the whole path (from IE)
  just_filename = File.basename(file_name)
  # replace all none alphanumeric, underscore or periods
  # with underscore
  just_filename.sub(/[^w\.\-]/, '_')
```

```
end
```

Deleting an Existing File

If you want to delete any existing file, it is quite simple. All that you need to do is write the following code:

```
def cleanup
  File.delete("#{RAILS_ROOT}/dirname/#{@filename}")
  if File.exist?("#{RAILS_ROOT}/dirname/#{@filename}")
  end
end
```

For a complete detail on **File** object, you need to go through our **Ruby Reference Manual**.

15. Sending Emails

ActionMailer is the Rails component that enables applications to send and receive emails. In this chapter, we will see how to send an email using Rails.

Let's start with creating an **emails** project using the following command.

```
C:\ruby> rails -d mysql emails
```

Here we are using **-d mysql** option to specify our interest to use MySQL database. We can specify any other database name like *oracle* or *postgres* using the **-d** option. By default, Rails uses **SQLite** database.

Setting Up the Database

Even though we are not using a database in our application, but Rails needs it to proceed. So let's perform these additional steps.

Given below is the way to create a database:

```
mysql> create database emails;
Query OK, 1 row affected (0.01 sec)

mysql> grant all privileges on emails.*
to 'root'@'localhost' identified by 'password';
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

To direct Rails to locate the database, edit the configuration file `~\upload\config\database.yml` and change the database name to `cookbook`. When you finish, it should look as follows:

```
development:
  adapter: mysql
  encoding: utf8
  database: emails
  username: root
```

```
password: password
host: localhost
test:
  adapter: mysql
  encoding: utf8
  database: emails
  username: root
  password: password
  host: localhost
production:
  adapter: mysql
  encoding: utf8
  database: emails
  username: root
  password: password
  host: localhost
```

ActionMailer – Configuration

Following are the steps you have to follow to complete your configuration before proceeding with the actual work.

Go to the config folder of your emails project and open the **environment.rb** file and add the following line at the bottom of this file.

```
ActionMailer::Base.delivery_method = :smtp
```

It informs the ActionMailer that you want to use the SMTP server. You can also set it as :sendmail if you are using a Unix-based operating system such as Mac OS X or Linux.

Add the following lines of code at the bottom of your environment.rb as well.

```
ActionMailer::Base.smtp_settings = {
  :address => "smtp.tutorialspoint.com",
  :port => 25,
  :domain => "tutorialspoint.com",
  :authentication => :login,
```

```
:user_name => "username",  
:password => "password",  
}
```

Replace each hash value with proper settings for your Simple Mail Transfer Protocol (SMTP) server. You can take this information from your Internet Service Provider if you already don't know. You don't need to change port number 25 and authentication type if you are using standard SMTP server.

You may also change the default email message format. If you prefer to send email in HTML instead of plain text format, add the following line to config/environment.rb as well:

```
ActionMailer::Base.default_content_type = "text/html"
```

ActionMailer::Base.default_content_type could be set to "text/plain", "text/html", and "text/enriched". The default value is "text/plain".

The next step is to create a mailer.

Generate a Mailer

Use the following command to generate a mailer as follows:

```
C:\ruby\> cd emails  
C:\ruby\emails> ruby script/generate mailer Emailer
```

It will create a file emailer.rb in the app/models directory. Check the content of this file as follows:

```
class Emailer < ActionMailer::Base  
end
```

Now let's create one method inside the ActionMailer::Base class as follows:

```
class Emailer < ActionMailer::Base  
  def contact(recipient, subject, message, sent_at = Time.now)  
    @subject = subject  
    @recipients = recipient  
    @from = 'no-reply@yourdomain.com'  
    @sent_on = sent_at  
    @body["title"] = 'This is title'
```

```

        @body["email"] = 'sender@yourdomain.com'
        @body["message"] = message
        @headers = {}
    end
end

```

The contact method has four parameters: a recipient, a subject, a message, and a sent_at, which defines when the email is sent. The method also defines six standard parameters that are a part of every ActionMailer method:

- @subject defines the e-mail subject.
- @body is a Ruby hash that contains values with which you can populate the mail template. You created three key-value pairs: title, email, and message
- @recipients is a list of the people to whom the message is being sent.
- @from defines who the e-mail is from.
- @sent_on takes the sent_at parameter and sets the timestamp of the e-mail.
- @headers is another hash that enables you to modify the e-mail headers. For example, you can set the MIME type of the e-mail if you want to send either plain text or HTML e-mail.

Creating the Controller

Now, we will create a controller for this application as follows:

```
C:\ruby\emails> ruby script/generate controller Emailer
```

Let's define a controller method **sendmail** in app/controllers/emailer_controller.rb, which will call the Model method to send an actual email as follows:

```

class EmailerController < ApplicationController
  def sendmail
    recipient = params[:email]
    subject = params[:subject]
    message = params[:message]
    Emailer.deliver_contact(recipient, subject, message)
    return if request.xhr?
  end
end

```

```

        render :text => 'Message sent successfully'
      end
    end
  end
end

```

To deliver e-mail using the mailer's contact method, you have to add **deliver_** to the beginning of the method name. You add a return if request.xhr?, so that you can escape to Rails Java Script (RJS) if the browser does not support JavaScript and then instruct the method to render a text message.

You are almost done except to prepare a screen from where you will get the user information to send email. Let's define one screen method index in the controller and then in the next section, we will define all the required views:

Add the following code in `emailer_controller.rb` file.

```

def index
  render :file => 'app\views\emailer\index.html.erb'
end

```

Defining Views

Define a view in `app\views\emails\index.html.erb`. This will be called as the default page for the application and will allow users to enter message and send the required email:

```

<h1>Send Email</h1>
<% form_tag :action => 'sendmail' do %>
  <p><label for="email_subject">Subject</label>:
  <%= text_field 'email', 'subject' %></p>
  <p><label for="email_recipient">Recipient</label>:
  <%= text_field 'email', 'recipient' %></p>
  <p><label for="email_message">Message</label><br/>
  <%= text_area 'email', 'message' %></p>
  <%= submit_tag "Send" %>
<% end %>

```

Apart from the above view, we need one more template, which will be used by the Emailer's contact method while sending message. This is just text with standard Rails `<%= %>` placeholders scattered throughout.

Just put the following code in the **app/views/contact.html.erb** file.

```
Hi!

You are having one email message from <%= @email %> with a title

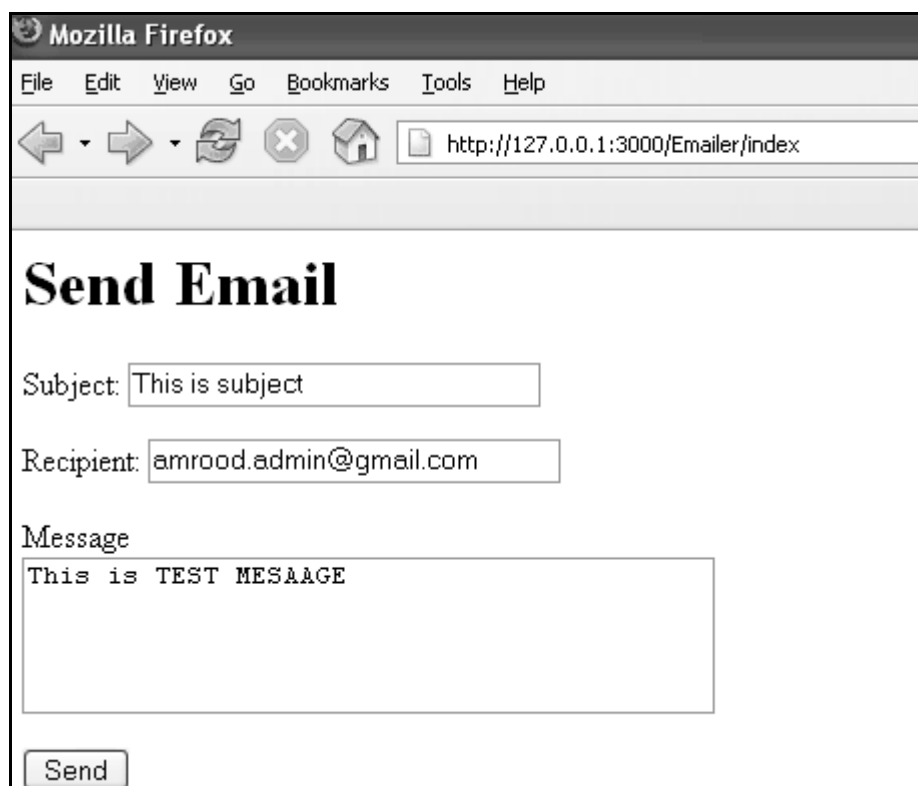
<%= @title %>
and following is the message:
<%= @message %>

Thanks
```

Rest for Testing

Before testing, make sure your machine is connected to the internet and your Email Server and the Webserver are up and running.

Now, test your application by using <http://127.0.0.1:3000/Emailer/index>. It displays the following screen and by using this screen, you will be able to send your message to anybody.



Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://127.0.0.1:3000/Emailer/index

Send Email

Subject:

Recipient:

Message

After sending a message, it will display the text message - "Message sent successfully".

Sending HTML Emails using Rails

To send mails as HTML, make sure your view (the .erb file) generates HTML and set the content type to html in your **emails/app/models/emailer.rb** file as follows:

```
class Emailer < ActionMailer::Base
  def contact(recipient, subject, message, sent_at = Time.now)
    @subject = subject
    @recipients = recipient
    @from = 'no-reply@yourdomain.com'
    @sent_on = sent_at
    @body["title"] = 'This is title'
    @body["email"] = 'sender@yourdomain.com'
    @body["message"] = message
    @headers = {content_type => 'text/html'}
  end
end
```

For a complete detail on **ActionMailer**, please look into the standard Ruby documentation.

16. RMagick

Rails provides bindings to *ImageMagick* and *GraphicsMagick*, which are popular and stable C libraries. The **RMagick** library provides the same interface against ImageMagick and GraphicsMagick, so it does not matter which one you use.

You can get RMagick by installing the `rmagick` gem on Unix or `rmagick-win32` gem on Windows. Let's install it on a Unix machine as follows:

```
$ gem install rmagick
```

The **RMagick** module comes along with class *Magick::Image*, which lets you resize images in four different methods:

- `resize(width, height)`
- `scale(width, height)`
- `sample(width, height)`
- `thumbnail(width, height)`

All these methods accept a pair integer values, corresponding to the width and height in pixels of the thumbnail you want.

Example

Here's an example that uses *resize()* method to resize the image. It takes the file **tmp.jpg** and makes a thumbnail of it 100 pixels wide by 100 pixels tall:

```
require 'rubygems'
require 'RMagick'

class ImageController < ApplicationController

  def createThumbnail
    width, height = 100, 100

    img = Magick::Image.read('tmp.jpg').first
    thumb = img.resize(width, height)

    # If you want to save this image use following
```

```

    # thumb.write("mythumbnail.jpg")

    # otherwise send it to the browser as follows
    send_data(thumb.to_blob, :disposition => 'inline',
              :type => 'image/jpg')

  end
end

```

Here are the steps to create a thumbnail:

- Here the class method *Image.read* receives an image filename as an argument and returns an array of Image objects. You obtain the first element of that array, which is obviously our *tmp.jpg* image.
- Next, we are calling method **resize** with the desired arguments, which is creating a thumbnail.
- Finally, we are directing this image to the browser. You can also use the method *thumb.write("mythumbnail.jpg")* to store this image locally at your machine.

Converting Image Formats

It is very easy to convert an image file from one format to another format. **RMagick** handles it very smartly. You can just read in the file and write it out with a different extension.

Example

The following example converts a JPEG file into a GIF file:

```

require 'rubygems'
require 'RMagick'

class ImageController < ApplicationController

  def changeFormat

    img = Magick::Image.read('tmp.jpg').first

    # If you want to save this image use following

```

```
# img.write("mythumbnail.gif")

# otherwise send it to the browser as follows
send_data(img.to_blob, :disposition => 'inline',
           :type => 'image/gif')

end

end
```

You can change an image to a format based on your requirement as follows:

```
img = Magick::Image.read("tmp.png").first
img.write("tmp.jpg")           # Converts into JPEG
img.write("tmp.gif")           # Converts into GIF
img.write("JPG:tmp")           # Converts into JPEG
img.write("GIF:tmp")           # Converts into GIF
```

17. HTTP Basic Authentication

Rails provides various ways of implementing authentication and authorization. But the simplest one is a new module, which has been added in Rails 2.0. This module is a great way to do API authentication over SSL.

To use this authentication, you will need to use SSL for traffic transportation. In our tutorial, we are going to test it without an SSL.

Let us start with our library example that we have discussed throughout the tutorial. We do not have much to do to implement authentication. We will add a few lines in blue in our **~library/app/controllers/book_controller.rb**:

Finally, your **book_controller.rb** file will look as follows:

```
class BookController < ApplicationController

  USER_ID, PASSWORD = "zara", "pass123"

  # Require authentication only for edit and delete operation
  before_filter :authenticate, :only => [ :edit, :delete ]

  def list
    @books = Book.find(:all)
  end
  def show
    @book = Book.find(params[:id])
  end
  def new
    @book = Book.new
    @subjects = Subject.find(:all)
  end
  def create
    @book = Book.new(params[:book])
    if @book.save
      redirect_to :action => 'list'
    end
  end
end
```

```

    else
      @subjects = Subject.find(:all)
      render :action => 'new'
    end
  end
end

def edit
  @book = Book.find(params[:id])
  @subjects = Subject.find(:all)
end

def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to :action => 'show', :id => @book
  else
    @subjects = Subject.find(:all)
    render :action => 'edit'
  end
end

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

def show_subjects
  @subject = Subject.find(params[:id])
end

private

def authenticate
  authenticate_or_request_with_http_basic do |id, password|
    id == USER_ID && password == PASSWORD
  end
end
end

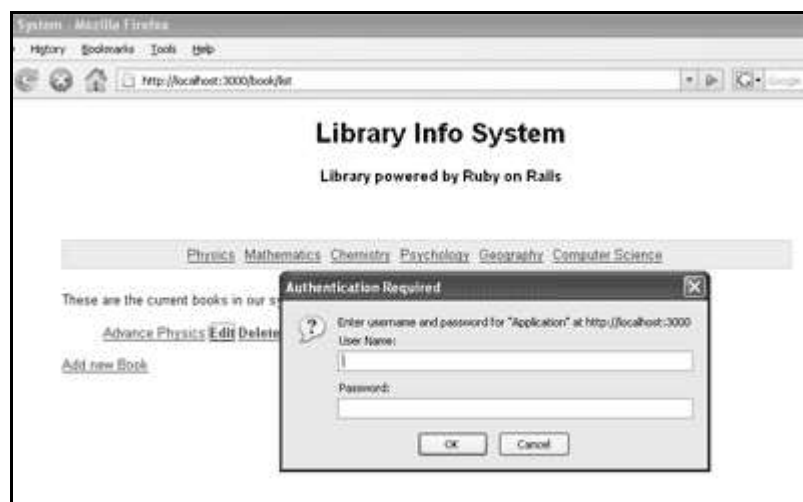
```

```
end
```

Let us explain these new lines:

- The first line is just to define the user ID and password to access various pages.
- In the second line, we have put *before_filter*, which is used to run the configured method *authenticate* before any action in the controller. A filter may be limited to specific actions by declaring the actions to include or exclude. Both options accept single actions (:only => :index) or arrays of actions (:except => [:foo, :bar]). So here we have put authentication for edit and delete operations only.
- Due to the second line, whenever you would try to edit or delete a book record, it will execute private *authenticate* method.
- A private *authenticate* method is calling *authenticate_or_request_with_http_basic* method, which comprises of a block and displays a dialogue box to ask for User ID and Password to proceed. If you enter a correct user ID and password then it will proceed, otherwise it would display 'access denied'.

Now, try to edit or delete any available record, to do so you would have to go through the authentication process using the following dialogue box.



18. Exception Handling

Execution and exception always go together. If you are opening a file that does not exist, then you need to handle this situation properly, or your program is considered to be of substandard quality.

The program stops if an exception occurs. Exceptions are used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.

Exception handling in Ruby on Rails is similar to exception handling in Ruby. Which means, we enclose the code that could raise an exception in a *begin/end* block and use *rescue* clauses to tell Ruby the types of exceptions we want to handle.

Syntax

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# Other exceptions
ensure
# Always will be executed
end
```

Everything from *begin* to *rescue* is protected. If an exception occurs during the execution of this block of code, control is passed to the block between *rescue* and *end*.

For each *rescue* clause in the *begin* block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is same as the type of the currently thrown exception, or is a superclass of that exception.

Where to Log Errors?

You have three options when an exception is thrown:

- Log to an internal log file (`logger.error`)
- Display an appropriate message to the user
- Redisplay the original page to continue

The error reporting to the application is done to a structure called a *flash*. The *flash* is a hash bucket to contain your message until the next request before being deleted automatically. You can access it with the `@flash` variable. Given below is the simplest form of using *logger* module to log error messages in an internal file.

```
begin
.....
rescue Exception => exc
  logger.error("Message for the log file #{exc.message}")
  flash[:notice] = "Store error message"
  redirect_to(:action => 'index')
end
```

To display all the messages from `@flash` in your view or layout (`.html.erb`), you can add the following:

```
<% @flash[:notice] -%>
<div id="notice"><%= @flash[:notice] %></div>
<% end -%>
```

19. Routes System

Rails parses the URL to determine the controller, action, and parameters for the request. With Rails routing, parts of the URL can specify additional parameters, and the entire routing process is under your control. Routing rules work the same on any web server.

The `config/routes.rb` file is at the heart of the Rails routing system. This file contains rules that try to match the URL path of a request and determine where to direct that request. The rules are tested in the order they are defined in the file. The first rule to match a request's URL path determines the fate of that request.

The routing system actually does two things:

- It maps requests to action methods inside the controllers.
- It writes URLs for you for use as arguments to methods like `link_to`, `redirect_to`, and `form_tag`.

Thus, the routing system knows how to turn a visitor's request URL into a controller/action sequence. It also knows how to manufacture URL strings based on your specifications.

Consider the following route installed by Rails when you generate your application:

```
map.connect ':controller/:action/:id'
```

This route states that it expects requests to consist of a **:controller** followed by an **:action** that in turn is fed some **:id**.

If you get an incoming request for "`http://localhost:3000/book/edit/2`", then it will map as follows:

```
params = { :controller => 'book',  
           :action     => 'edit',  
           :id         => '2'  
}
```

Thus the default routing (if you don't modify the routing rules) is:

```
http://<base-url>/<controller>/<action>/<id>
```

A URL like `http://www.example.com/book/update/20` calls the **update** method (the action) in the **BooksController** class (the controller) with an **id** parameter set to the value 20.

The following code block will set up *book* as the default controller if no other is specified. It means visiting '/' would invoke the **book** controller.

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id', :controller => 'book'
end
```

You can also define a default action if no action is specified in the given URL:

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id',
             :action => 'edit', :controller => 'book'
end
```

Now, you can all edit methods inside the book controller to edit book with ID as 20 as follows:

```
http://localhost:3000/2
```

Route Priority

Routes have priority defined by the order of appearance of the routes in the routes.rb file. The priority goes from top to bottom.

The last route in that file is at the lowest priority and will be applied last. If no route matches, 404 is returned.

Modifying the Default Route

You can change the default route as per your requirement. In the following example, we are going to interchange *controller* and *action* as follows:

```
# Install the default route as the lowest priority.
map.connect ':action/:controller/:id'
```

Now, to call *action* from the given *controller*, you would have to write your URL as follows:

```
http://localhost:3000/action/controller/id
```

It's not particularly logical to put *action* and *controller* in such sequence. The original default (the default default) route is better and recommended.

The Ante-Default Route

The 'ante-default' route looks as follows:

```
map.connect ':controller/:action/:id.:format'
```

The **..format** at the end matches a literal dot and a wildcard "format" value after the **id** field. That means it will match, for example, a URL like this:

```
http://localhost:3000/book/show/3.xml
```

Here, inside the controller action, your **params[:format]** will be set to xml.

The Empty Route

The empty route is sort of the opposite of the default route. In a newly generated routes.rb file, the empty route is commented out, because there's no universal or reasonable default for it. You need to decide what this *nothing* URL should do for each application you write.

Here are some examples of fairly common empty route rules:

```
map.connect '', :controller => "main", :action => "welcome"
map.connect '', :controller => "main"
```

Here is the explanation of the above rules:

- The first one will search for welcome action inside main controller even if you type just *http://localhost:3000*.
- That last one will connect to *http://localhost:3000/main/index*. Here *index* is the default action when there's none specified.

Rails 2.0 introduces a mapper method named *root*, which becomes the proper way to define the empty route for a Rails application, like this:

```
map.root :controller => "homepage"
```

Defining the empty route gives people something to look at when they connect to your site with nothing but the domain name.

Named Routes

As you continue developing your application, you will probably have a few links that you use throughout your application. For example, you will probably often be putting a link back to the main listings page. Instead of having to add the following line throughout your application, you can instead create a named route that enables you to link to a shorthand version of that link:

```
link_to 'Home', :controller => 'classified', :action => 'list'
```

You can define named routes as follows. Here instead of using *connect*, you are using a unique name that you can define. In this case, the route is called *home*. The rest of the route looks similar to the others you have created.

```
map.home '', :controller => 'classified', :action => 'list'
```

Now, you can use this in the controllers or views as follows:

```
<%= link_to 'Back', home_url %>
```

Here, instead of listing the *:controller* and *:action* to which you will be linking, you are instead putting the name of the route followed by *_url*. Your user shouldn't notice any difference. Named routing is merely a convenience for the Rails developer to save some typing. The above case can be written without the named route as follows:

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

Pretty URLs

Routes can generate pretty URLs. For example:

```
map.connect 'articles/:year/:month/:day',
  :controller => 'articles',
  :action     => 'find_by_date',
  :year       => /\d{4}/,
  :month      => /\d{1,2}/,
  :day        => /\d{1,2}/

  # Using the route above, the url below maps to:
  # params = {:year => '2005', :month => '11', :day => '06'}
  # http://localhost:3000/articles/2005/11/06
```

To obtain more detail on Routes, please go through **ActionController::Routing**.

20. Unit Testing

Introduction

Before proceeding, let's have a quick look at a few definitions:

- The **Tests** - They are test applications that produce consistent result and prove that a Rails application does what it is expected to do. Tests are developed concurrently with the actual application.
- The **Assertion** - This is a one line of code that evaluates an object (or expression) for expected results. For example - Is this value = that value? Is this object nil?
- The **Test Case** - This is a class inherited from `Test::Unit::TestCase` containing a testing *strategy* comprised of contextually related tests.
- The **Test Suite** - This is a collection of test cases. When you run a test suite, it will, in turn, execute each test that belongs to it.

Rails Testing

When you run the helper script *script/generate* to create *controllers* and *models*, Rails generate a framework for unit and functional tests. You can get pretty good test coverage by filling in the framework with tests for the functionality you write. There are two important points to test in a Rails application:

- Testing the Models
- Testing the Controllers

This tutorial will cover both the testings in brief. So let's create one *testapp* to understand the concept.

```
C:\ruby> rails -d mysql testapp
```

Database Setup

Till now, we have used only Rails application's development database, but now you need to make sure that the testing database is also created and appropriate sections of your `config/database.yml` file are set up correctly.

Let's create development and testing databases as follows:

```
mysql> create database testapp_test;
Query OK, 1 row affected (0.01 sec)

mysql> create database testapp_development;
Query OK, 1 row affected (0.01 sec)

mysql> use testapp_test;
Database changed

mysql> grant all privileges on testapp_test.*
      to 'root'@'localhost' identified by 'password';
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

Configuring database.yml:

Configure your config/database.yml as follows:

```
development:
  adapter: mysql
  encoding: utf8
  database: testapp_development
  username: root
  password: password
  host: localhost
test:
  adapter: mysql
  encoding: utf8
  database: testapp_test
  username: root
```



```

password: password
host: localhost
production:
  adapter: mysql
  encoding: utf8
  database: testapp_production
  username: root
  password: password
  host: localhost

```

Generate Migration

Assume you have a table containing books, including their titles, price, and a small description. The following migration sets up this table:

```
testapp > ruby script/generate migration books
```

Now modify the testapp/db/migrate/20080616170315_books.rb file as follows:

```

class Books < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.string    :title, :limit => 32, :null => false
      t.float     :price
      t.text     :description
      t.timestamp :created_at
    end
  end
  def self.down
    drop_table :books
  end
end

```

Now run the migration as follows:

```
testapp > rake db:migrate
```

This will create **books** table in `testapp_development` database. Thereafter, we need to set up your test database using *rake* command as follows:

```
C:\ruby\testapp > rake db:test:clone_structure
```

This will clone the *testapp_development* database into *testapp_test* database. It means whatever you have in the development database, now you will have the same data in the test database as well.

Testing Models

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

```
testapp > ruby script/generate model Book

exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/book.rb
create  test/unit/book_test.rb
create  test/fixtures/books.yml
create  db/migrate
create  db/migrate/20080616164236_create_books.rb
```

As you write code in the model classes, you'll write corresponding tests in these files. So let's create two test book records using *YAML* in *test/fixtures/books.yml* as follows:

```
perl_cb:
  id: 1
  title: 'Ruby Tutorial'
  price: 102.00
  description : 'This is a nice Ruby tutorial'
java_cb:
  id: 2
  title: 'Java Programming'
  price: 62.00
```

```
description : 'Java Programming for the beginners'
```

Now let's replace the existing code in book unit test file test/unit/book_test.rb with the following code:

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

class BookTest < ActiveSupport::TestCase
  fixtures :books

  def test_book

    perl_book = Book.new :title => books(:perl_cb).title,
                        :price => books(:perl_cb).price,
                        :description => books(:perl_cb).description,
                        :created_at => books(:perl_cb).created_at

    assert perl_book.save

    perl_book_copy = Book.find(perl_book.id)

    assert_equal perl_book.title, perl_book_copy.title

    perl_book.title = "Ruby Tutorial"

    assert perl_book.save
    assert perl_book.destroy
  end
end
```

Finally, run the test method as follows:

```
testapp > ruby test/unit/book_test.rb
```

Here's the output of running the successful test case:

```
testapp > ruby test/unit/book_test_crud.rb
```

```
Loaded suite ./test/unit/book_test
Started
.
Finished in 0.0625 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

Let's analyze what happened here:

1. The BookTest method starts off by creating a new Book object using the title and other fields from the first record in the text fixture/books.yml. The resulting object is stored in the perl_book instance variable.
2. The first assertion tests that saving the Book object was successful.
3. Next, the book object is retrieved using the *find* method and stored in another instance variable named perl_book_copy. The success of this retrieval is tested in the next assertion, which compares the titles of both book objects. At this point, we've tested the ability to create and read a database record.
4. The solution tests updating by assigning a new title to the object stored in perl_book and then asserts that saving the change is successful.
5. Finally, the ability to destroy a Book object is tested.

This is how we can test our Rails Models.

Testing the Controllers

Controller testing is also known as **functional testing**. Functional testing tests the following type of functionalities of the controllers:

- Is the response redirected as expected?
- Is the expected template rendered?
- Is the routing as expected?
- Does the response contain the expected tags?

Rails framework supports five types of requests:

- get
- post
- put
- head

- delete

To write a functional test, you need to simulate any of the five HTTP request types that your controller will process.

Request type "get" and "post" are the most commonly used in controller testing. All these methods take four arguments:

- The action of a controller
- An optional hash of request parameters
- An optional session hash
- An optional flash hash

In this tutorial, we will see how to use **get** method to test our controller. You can test rest of the methods in similar way.

When you generate a *controller* with generate, Rails creates a functional test script for the controller as follows:

```
testapp > ruby script/generate controller Book
exists app/controllers/
exists app/helpers/
create app/views/book
exists test/functional/
create app/controllers/book_controller.rb
create test/functional/book_controller_test.rb
create app/helpers/book_helper.rb
```

As you write code in the controller classes, you'll write corresponding tests in these files. Before that, let's define our controller functions *list*, *show*, and *search* inside **app/controllers/book_controller.rb** as follows:

```
class BookController < ApplicationController
  def list
    @book_pages, @books = paginate :books, :per_page => 10
  end

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

```

def search
  @book = Book.find_by_title(params[:title])
  if @book
    redirect_to :action => 'show', :id => @book.id
  else
    flash[:error] = 'No such book available'
    redirect_to :action => 'list'
  end
end
end
end

```

NOTE: You would need two views templates for **show** and *list* method. You can define those views and test them, but right now, we will proceed without defining those views.

Now let's reuse our test fixture which is in the **test/fixtures/books.yml** file as follows:

```

perl_cb:
  id: 1
  title: 'Ruby Tutorial'
  price: 102.00
  description : 'This is a nice Ruby tutorial'
java_cb:
  id: 2
  title: 'Java Programming'
  price: 62.00
  description : 'Java Programming for the beginners'

```

Add the following *test_search_book* and *test_search_not_found* methods to *test/functional/book_controller_test.rb* to test the functionality of the Book Controller's search action.

```

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

# Re-raise errors caught by the controller.
class BookController

```

```

    def rescue_action(e)
      raise e
    end
  end
end

class BookControllerTest < Test::Unit::TestCase
  fixtures :books
  def setup
    @controller = BookController.new
    @request     = ActionController::TestRequest.new
    @response    = ActionController::TestResponse.new
  end

  def test_search_book
    get :search, :title => 'Ruby Tutorial'
    assert_not_nil assigns(:book)
    assert_equal books(:perl_cb).title, assigns(:book).title
    assert_valid assigns(:book)
    assert_redirected_to :action => 'show'
  end

  def test_search_not_found
    get :search, :title => 'HTML Tutorial'
    assert_redirected_to :action => 'list'
    assert_equal 'No such book available', flash[:error]
  end
end
end

```

Now run your test cases as follows:

```
testapp > ruby test/functional/book_controller_test.rb
```

It gives the following output:

```
Loaded suite test/functional/book_controller_test
```

```
Started
..
Finished in 0.422 seconds.

2 tests, 7 assertions, 0 failures, 0 errors
```

Let's analyze what has happened here:

- The *setup* method is a default method to create controller, request, and response objects. They would be used by Rails internally.
- The first test method *test_search_book* generates a **get** request to the search action, passing in a *title* parameter.
- The next two assertions verify that a *Book* object was saved in an instance variable called *@book* and that the object passes any Active Record validations that might exist.
- The final assertion inside first method tests that the request was redirected to the controller's show action.
- The second test method, *test_search_not_found*, performs another *get* request but passes in an invalid title.
- The first assertions test that a redirect to the *list* action was issued.
- If the proceeding assertions passed, there should be a message in the *flash* hash which you can test with *assert_equal*.

To obtain more information on Assertions, please refer [Rails Standard Documentation](#).

Using Rake for Testing

You can use **rake** utility to test your applications. Given below is a list of few important commands.

- **\$rake test** - Test all unit tests and functional tests (and integration tests, if they exist).
- **\$rake test:functionals** - Run all functional tests.
- **\$rake test:units** - Run all unit tests.
- **\$rake test:integration** - Run all integration tests.
- **\$rake test:plugins** - Run all test in *./vendor/plugins/**/test*.
- **\$rake test:recent** - Run tests for models and controllers that have been modified in the last 10 minutes:

- **\$rake test:uncommitted** - For projects in Subversion, run tests for the changes that took place in the models and controllers since the last commit.

21. Quick Reference Guide

Here we have listed all the important functions, scripts, validations, etc.

- **Ruby on Rails - Rake Utility**
- **Ruby on Rails - The Scripts**
- **Ruby on Rails - The Plugins**
- **Ruby on Rails - The Generators**
- **Ruby on Rails - Model Relations**
- **Ruby on Rails - Controller Methods**
- **Ruby on Rails - Layouts**
- **Ruby on Rails - Render**
- **Ruby on Rails - HTML Forms**
- **Ruby on Rails - RXML**
- **Ruby on Rails - RHTML**
- **Ruby on Rails - HTML Links**
- **Ruby on Rails - Session and Cookies**
- **Ruby on Rails - User Input Validations**
- **Ruby on Rails - Maths Functions**
- **Ruby on Rails - Finders**
- **Ruby on Rails - Nested with-scope**
- **Ruby on Rails - Callback Functions**

Ruby on Rails – Rake Utility

Rake is a utility similar to **make** in Unix. You can say Rake is the make of ruby - the RubyMake. Rails defines a number of tasks to help you.

Here is a list of various important commands supported by Rake:

- **rake db:fixtures:load** - Load fixtures into the current environment's database. Load specific fixtures using FIXTURES=x, y.
- **rake db:migrate** - Migrate the database through scripts in db/migrate. Target specific version with VERSION=x.

- **rake db:schema:dump** - Create a db/schema.rb file that can be portably used against any DB supported by AR.
- **rake db:schema:load** - Load a schema.rb file into the database.
- **rake db:sessions:clear** - Clear the sessions table.
- **rake db:sessions:create** - Creates a sessions table for use with CGI::Session::ActiveRecordStore.
- **rake db:structure:dump** - Dump the database structure to a SQL file.
- **rake db:test:clone** - Recreate the test database from the current environment's database schema.
- **rake db:test:clone_structure** - Recreate the test databases from the development structure.
- **rake db:test:prepare** - Prepare the test database and load the schema.
- **rake db:test:purge** - Empty the test database.
- **rake doc:app** - Build the app HTML Files.
- **rake doc:clobber_app** - Remove rdoc products.
- **rake doc:clobber_plugins** - Remove plugin documentation.
- **rake doc:clobber_rails** - Remove rdoc products.
- **rake doc:plugins** - Generate documentation for all installed plugins.
- **rake doc:rails** - Build the rails HTML Files.
- **rake doc:reapp** - Force a rebuild of the RDOC files
- **rake doc:reraails** - Force a rebuild of the RDOC files
- **rake log:clear** - Truncate all *.log files in log/ to zero bytes
- **rake rails:freeze:edge** - Lock this application to latest Edge Rails. Lock a specific revision with REVISION=X.
- **rake rails:freeze:gems** - Lock this application to the current gems (by unpacking them into vendor/rails)
- **rake rails:unfreeze** - Unlock this application from freeze of gems or edge and return to a fluid use of system gems
- **rake rails:update** - Update both scripts and public/javascripts from Rails.
- **rake rails:update:javascripts** - Update your javascripts from your current rails install.
- **rake rails:update:scripts** - Add new scripts to the application script/ directory.

- **rake stats** - Report code statistics (KLOCs, etc) from the application.
- **rake test** - Test all units and functionals
- **rake test:functionals** - Run tests for functionalsdb:test:prepare
- **rake test:integration** - Run tests for integrationdb:test:prepare
- **rake test:plugins** - Run tests for pluginsenvironment
- **rake test:recent** - Run tests for recentdb:test:prepare
- **rake test:uncommitted** - Run tests for uncommitteddb:test:prepare
- **rake test:units** - Run tests for unitsdb:test:prepare
- **rake tmp:cache:clear** - Clear all files and directories in tmp/cache
- **rake tmp:clear** - Clear session, cache, and socket files from tmp/
- **rake tmp:create** - Create tmp directories for sessions, cache, and sockets
- **rake tmp:sessions:clear** - Clear all files in tmp/sessions
- **rake tmp:sockets:clear** - Clear all ruby_sess.* files in tmp/sessions.

Ruby on Rails – The Scripts

Ruby provides many useful scripts, which you use during your project implementation. Here are few scripts which you will use most frequently.

Script	Description
script/about	Gives information about environment.
script/breakpointer	Starts the breakpoint server.
script/console	Interactive Rails Console.
script/destroy	Deletes files created by generators.
script/generate	Used by generators.
script/runner	Executes a task in the rails context.
script/server	Launches the development server.

script/performance/profile	Profiles an expansive method.
script/performance/benchmark	Benchmarks different methods.

Script Example

Here is an example to call a script. The following script will start the development server.

```
C:\ruby> ruby script/server
```

Ruby on Rails – The Plugins

Ruby provides many useful plug-ins that you can use in your Rails applications. The following table lists a few scripts that will help you in dealing with plug-ins.

Plugin	Description
script/plugin discover	Discovers plugin repositories.
script/plugin list	Lists all available plugins.
script/plugin install where	Installs the "where" plugin.
script/plugin install -x where	Installs the "where" plugin as SVN external.
script/plugin update	Updates installed plugins.
script/plugin source	Adds a source repository.
script/plugin unsource	Removes a source repository.
script/plugin sources	Lists source repositories.

Ruby on Rails – The Generators

Ruby provides a script called **Generator**. This script can be used to generate many useful items in Rails. The most important generators are listed below.

Generator	Description
<code>script/generate model ModelName</code>	Generates Active Records.
<code>script/generate controller ListController show edit</code>	Generates Controller.
<code>script/generate scaffold ModelName ControllerName</code>	Generates Scaffold.
<code>script/generate migration AddNewTable</code>	Generates Table to migrate.
<code>script/generate plugin PluginName</code>	Generates Plugin.
<code>script/generate integration_test TestName</code>	Generates Integ Test.
<code>script/generate session_migration</code>	Generates Session Migration.

Given below is a list of options that can be used along with generators:

- **-p, --pretend** Run but do not make any changes.
- **-f, --force** Overwrite files that already exist.
- **-s, --skip** Skip files that already exist.
- **-q, --quite** Suppress normal output.
- **-t, --backtrace** Debugging: show backtrace on errors.
- **-h, --help** Show this help message.
- **-c, --svn** Modify files with subversion.

Ruby on Rails – Model Relations

Model Creation

```
Model.new    # creates a new empty model
Model.create( :field => 'value', :other_field => 42 )
# creates an object with the passed parameters and saves it
```

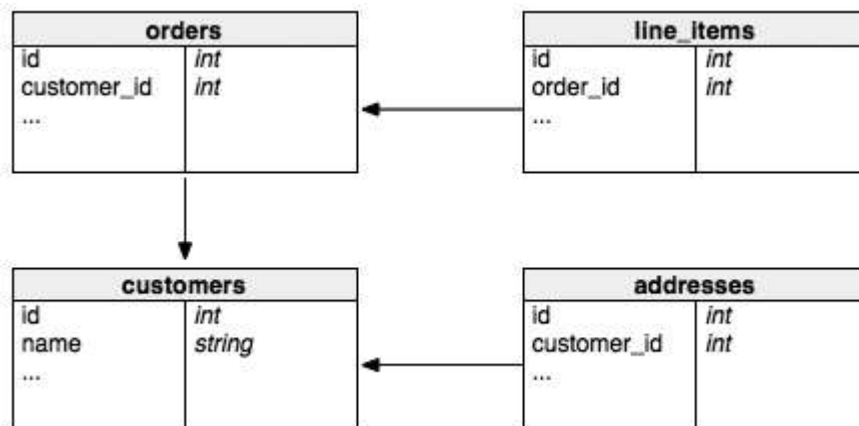
```

Model.find_or_create_by_field( value )
# searches for a record where "field = value", creates
# a new record if not found
User.find_or_create_by_name_and_email( 'ramjoe', 'ram@example.com')

```

Model Relations

There are four ways of associating models: `has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`. Assuming the following four entities:

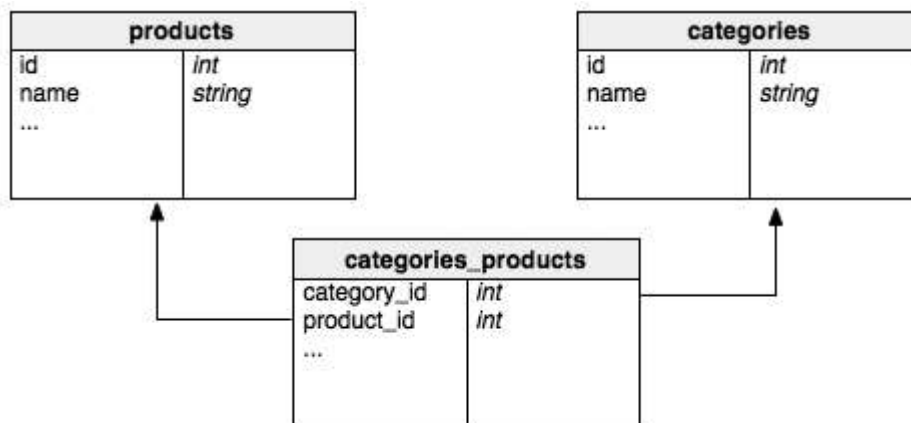


```

def Order < ActiveRecord::Base
  has_many :line_items
  belongs_to :customer
end
def LineItem < ActiveRecord::Base
  belongs_to :order
end
def Customer < ActiveRecord::Base
  has_many :orders
  has_one :address
end
def Address < ActiveRecord::Base
  belongs_to :customer
end

```

Consider the following relationship:



```

def Category < ActiveRecord::Base
  has_and_belongs_to_many :products

end

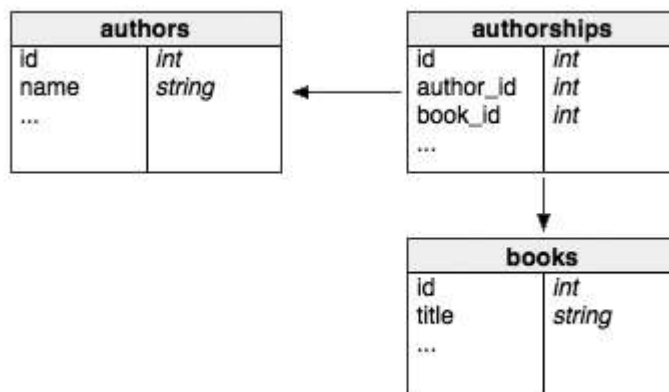
def Product < ActiveRecord::Base
  has_and_belongs_to_many :categories

end

```

Association Join Models

Consider the following relationship. It depicts how we can use joins while defining a relationship.



```

class Author < ActiveRecord::Base

```



```

    has_many :authorships
    has_many :books, :through => :authorships
  end
class Authorship < ActiveRecord::Base
  belongs_to :author
  belongs_to :book
end
class Book < ActiveRecord::Base
  has_one :authorship
end
@author = Author.find :first
# selects all books that the author's authorships belong to.
@author.authorships.collect { |a| a.book }
selects all books by using the Authorship join model
@author.books

```

Ruby on Rails – Controller Methods

Each public method in a controller is callable by the (standard) URL scheme /controller/action.

```

class WorldController < ApplicationController
  def hello
    render :text => 'Hello world'
  end
end

```

Parameters are stored in the params hash:

```

/world/hello/1?foo=bar
id = params[:id]      # 1
foo = params[:foo]    # bar

```

Instance variables defined in the controllers methods are available to the corresponding view templates:

```

def show
  @person = Person.find( params[:id])
end

```

```
end
```

Distinguish the type of response accepted:

```
def index
  @posts = Post.find :all
  respond_to do |type|
    type.html # using defaults, which will render weblog/index.rhtml
    type.xml { render :action => "index.rxml" }
    type.js   { render :action => "index.rjs" }
  end
end
```

Ruby on Rails – Layouts

A layout defines the surroundings of an HTML page. It's the place to define the common look and feel of your final output. Layout files reside in `app/views/layouts`.

The process involves defining a layout template and then letting the controller know that it exists and is available for use. First, let's create the template.

Add a new file called `standard.rhtml` to `app/views/layouts`. You let the controllers know what template to use by the name of the file, so following a same naming scheme is advised.

Add the following code to the new `standard.rhtml` file and save your changes:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
<meta http-equiv="Content-Language" content="en-us" />
<title>Library Info System</title>
<%= stylesheet_link_tag "style" %>
</head>
<body id="library">
```

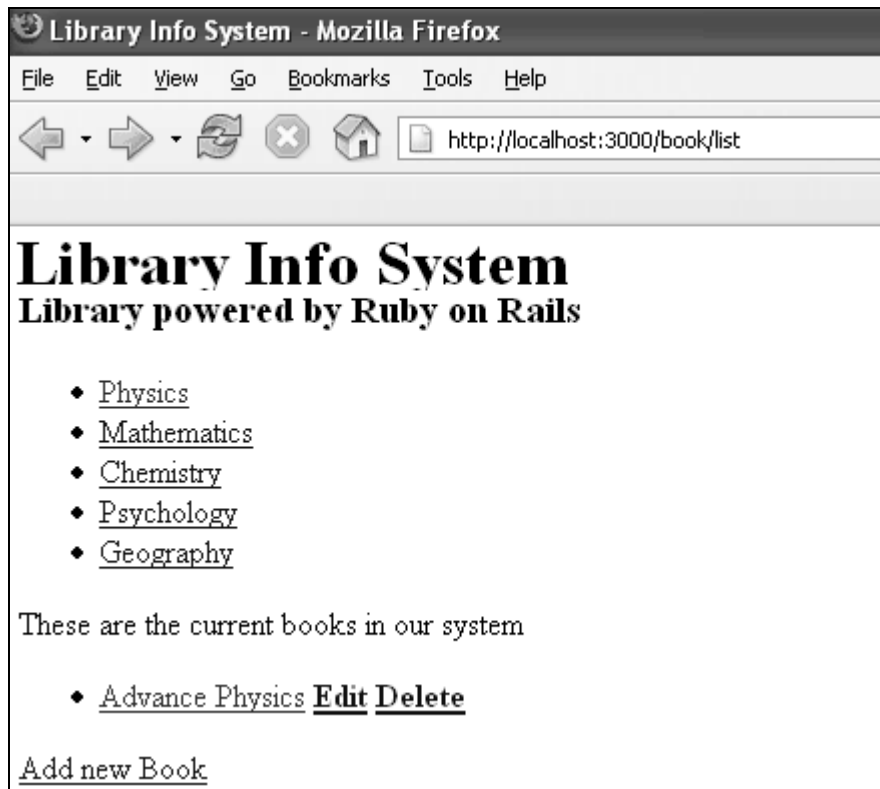
```
<div id="container">
<div id="header">
<h1>Library Info System</h1>
<h3>Library powered by Ruby on Rails</h3>
</div>
<div id="content">
<%= yield -%>
</div>
<div id="sidebar"></div>
</div>
</body>
</html>
```

Everything you just added are standard HTML elements except two lines. The **stylesheet_link_tag** helper method outputs a stylesheet <link>. In this instance, we are linking the style.css stylesheet. The **yield** command lets Rails know that it should put the RHTML for the method called here.

Now open **book_controller.rb** and add the following line just below the first line:

```
class BookController < ApplicationController
  layout 'standard'
  def list
    @books = Book.find(:all)
  end
  .....
```

It instructs the controller that we want to use a layout available in the standard.rhtml file. Now, try browsing books that will produce the following screen.



Adding a Stylesheet

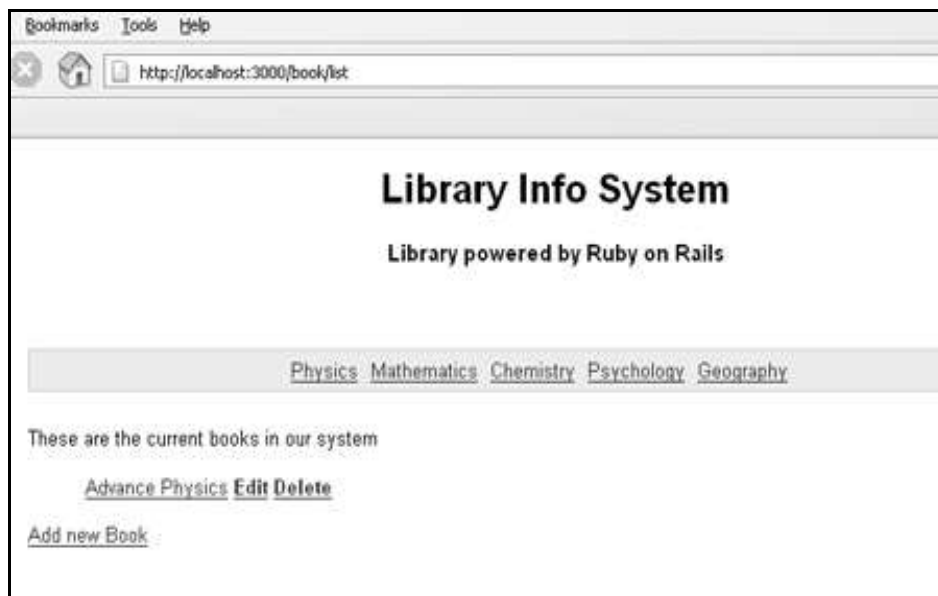
Till now, we have not created any stylesheet, so Rails is using the default stylesheet. Now let's create a new file called style.css and save it in /public/stylesheets. Add the following code to this file.

```
body {
  font-family: Helvetica, Geneva, Arial, sans-serif;
  font-size: small;
  font-color: #000;
  background-color: #fff;
}
a:link, a:active, a:visited {
  color: #CD0000;
}
input {
  margin-bottom: 5px;
}
p {
```

```
    line-height: 150%;
  }
  div#container {
    width: 760px;
    margin: 0 auto;
  }
  div#header {
    text-align: center;
    padding-bottom: 15px;
  }
  div#content {
    float: left;
    width: 450px;
    padding: 10px;
  }
  div#content h3 {
    margin-top: 15px;
  }
  ul#books {
    list-style-type: none;
  }
  ul#books li {
    line-height: 140%;
  }
  div#sidebar {
    width: 200px;
    margin-left: 480px;
  }
  ul#subjects {
    width: 700px;
    text-align: center;
    padding: 5px;
    background-color: #ececec;
```

```
border: 1px solid #ccc;
margin-bottom: 20px;
}
ul#subjects li {
display: inline;
padding-left: 5px;
}
```

Now refresh your browser and see the difference:



Ruby on Rails – Render

Usually the view template with the same name as the controller method is used to render the results.

Action

```
# The default. Does not need to be specified
# in a controller method called "some_action"
render :action => 'some_action'
render :action => 'another_action', :layout => false
render :action => 'some_action', :layout => 'another_layout'
```

Partial

Partials are stored in files called "_subformname" (_error, _subform, _listitem).

```
render :partial => 'subform'
render :partial => 'error', :status => 500
render :partial => 'subform', :locals =>
    { :variable => @other_variable }
render :partial => 'listitem', :collection => @list
render :partial => 'listitem', :collection => @list,
    :spacer_template => 'list_divider'
```

Template

It is more like rendering an action, but it finds the template based on the template root (app/views).

```
# renders app/views/weblog/show
render :template => 'weblog/show'
```

File

```
render :file => '/path/to/some/file.rhtml'
render :file => '/path/to/some/filenotfound.rhtml',
    status => 404, :layout => true
```

Text

```
render :text => "Hello World"
render :text => "This is an error", :status => 500
render :text => "Let's use a layout", :layout => true
render :text => 'Specific layout', :layout => 'special'
```

Inline Template

Uses ERb to render the "miniature" template.

```
render :inline => "<%= 'hello , ' * 3 + 'again' %>"
render :inline => "<%= 'hello ' + name %>",
```

```
:locals => { :name => "david" }
```

Nothing

```
render :nothing  
render :nothing, :status => 403    # forbidden
```

RJS

```
def refresh  
  render :update do |page|  
    page.replace_html 'user_list', :partial => 'user',  
                                   :collection => @users  
    page.visual_effect :highlight, 'user_list'  
  end  
end
```

Change the content-type

```
render :action => "atom.rxml",  
       :content_type => "application/atom+xml"
```

Ruby on Rails – HTML Forms

Form

To create a form tag with the specified action and with POST request, use the following syntax:

```
<% form_tag :action => 'update', :id => @some_object do %>  
  Ruby Block  
<% end %>  
  
or  
  
<% form_tag( { :action => :save, }, { :method => :post }) do %>
```


Ruby Block

```
<% end %>
```

Use :multipart => true to define a MIME-multipart form (for file uploads).

```
<% form_tag( { :action => 'upload'}, :multipart => true ) do %>
```

Ruby Block

```
<% end %>
```

File Upload

Define a multipart form in your view:

```
<% form_tag( { :action => 'upload' }, :multipart => true ) do %>
```

```
  Upload file: <%= file_field( "form", "file" ) %>
```

```
  <br />
```

```
  <%= submit_tag( "Upload file" ) %>
```

```
<% end %>
```

Handle the upload in the controller:

```
def upload
```

```
  file_field = @params['form']['file'] rescue nil
```

```
  # file_field is a StringIO object
```

```
  file_field.content_type # 'text/csv'
```

```
  file_field.full_original_filename
```

```
  ...
```

```
end
```

Text Fields

To create a text field, use the following syntax:

```
<%= text_field :modelname, :attribute_name, options %>
```

Have a look at the following example:

```
<%= text_field "person", "name", "size" => 20 %>
```

It will generate the following code:

```
<input type="text" id="person_name" name="person[name]"
      size="20" value="<%= @person.name %>" />
```

To create hidden fields, use the following syntax;

```
<%= hidden_field ... %>
```

To create password fields, use the following syntax;

```
<%= password_field ... %>
```

To create file upload fields, use the following syntax;

```
<%= file_field ... %>
```

Text Area

To create a text area, use the following syntax:

```
<%= text_area ... %>
```

Have a look at the following example:

```
<%= text_area "post", "body", "cols" => 20, "rows" => 40%>
```

It will generate the following code:

```
<textarea cols="20" rows="40" id="post_body" name="post[body]">
<%={@post.body}%>
</textarea>
```

Radio Button

To create a Radio Button, use the following syntax:

```
<%= radio_button :modelname, :attribute, :tag_value, options %>
```

Have a look at the following example:

```
radio_button("post", "category", "rails")
radio_button("post", "category", "java")
```

It will generate the following code:

```
<input type="radio" id="post_category" name="post[category]"
      value="rails" checked="checked" />
<input type="radio" id="post_category" name="post[category]"
      value="java" />
```

Checkbox Button

To create a Checkbox Button use the following syntax:

```
<%= check_box :modelname, :attribute,options,on_value,off_value%>
```

Have a look at the following example:

```
check_box("post", "validated")
```

It will generate the following code:

```
<input type="checkbox" id="post_validate" name="post[validated]"
      value="1" checked="checked" />
<input name="post[validated]" type="hidden" value="0" />
```

Let's check another example:

```
check_box("puppy", "gooddog", {}, "yes", "no")
```

It will generate the following code:

```
<input type="checkbox" id="puppy_gooddog" name="puppy[gooddog]"
      value="yes" />
<input name="puppy[gooddog]" type="hidden" value="no" />
```

Options

To create a dropdown list, use the following syntax:

```
<%= select :variable,:attribute,choices,options,html_options%>
```

Have a look at the following example:

```
select("post", "person_id",
      Person.find(:all).collect {|p| [ p.name, p.id ] })
```

This could generate the following code. It depends on what value is available in your database.

```
<select name="post[person_id]">
  <option value="1">David</option>
  <option value="2">Sam</option>
  <option value="3">Tobias</option>
</select>
```

Date Time

Following is the syntax to use data and time:

```
<%= date_select :variable, :attribute, options %>
<%= datetime_select :variable, :attribute, options %>
```

Following are examples of usage:

```
<%=date_select "post", "written_on"%>
<%=date_select "user", "birthday", :start_year => 1910%>
<%=date_select "user", "cc_date", :start_year => 2005,
                                :use_month_numbers => true,
                                :discard_day => true,
                                :order => [:year, :month]%>
<%=datetime_select "post", "written_on"%>
```

Ruby on Rails – RXML

Rails provide two classes to create XML markup and data structures.

- **Builder::XmlMarkup:** This Generates XML markup notation.
- **Builder::XmlEvents:** This Generates XML events (i.e., SAX-like).

Builder::XmlMarkup

This class can be used to create XML markup easily. All methods sent to an XmlMarkup object will be translated to the equivalent XML markup. Any method with a block will be treated as an XML markup tag with nested markup in the block.

Assuming `xm` is an `XmlMarkup` object, we have picked up one example from Rails standard documentation site. Here commented part has been generated from the corresponding Rails statement.

Example

```
xm.em("emphasized")
# => <em>emphasized</em>

xm.em { xmm.b("emp & bold") }
# => <em><b>emp & bold</b></em>

xm.a("A Link", "href"=>"http://onestepback.org")
# => <a href="http://onestepback.org">A Link</a>

xm.div { br }
# => <div><br/></div>

xm.target("name"=>"compile", "option"=>"fast")
# => <target option="fast" name="compile">
# NOTE: order of attributes is not specified.

xm.instruct!
# <?xml version="1.0" encoding="UTF-8"?>

xm.html {
  xm.head {
    xm.title("History")
    # <title>History</title>
  }
  xm.body {
    xm.comment! "HI"
    xm.h1("Header")
    xm.p("paragraph")
  }
}
```

Builder::XmlEvents

This class can be used to create a series of SAX-like XML events (e.g. `start_tag`, `end_tag`) from the markup code.

XmlEvent objects are used in a way similar to the XmlMarkup objects, except that a series of events are generated and passed to a handler rather than generating character-based markup.

Example

```
xe = Builder::XmlEvents.new(handler)
xe.title("HI")
# This sends start_tag/end_tag/text messages to the handler.
```

XML Event Handler

The handler object must expect the following events.

- **start_tag(tag, attrs):** Announces that a new tag has been found. *tag* is the name of the tag and *attrs* is a hash of attributes for the tag.
- **end_tag(tag):** Announces that an end tag for tag has been found.
- **text(text):** Announces that a string of characters (text) has been found. A series of characters may be broken up into more than one text call, so the client cannot assume that a single callback contains all the text data.

Ruby on Rails – RHTML

RHTML is HTML mixed with Ruby using HTML tags. All of Ruby is available for programming along with HTML.

Following is the syntax of using Ruby with HTML:

```
<% %>    # executes the Ruby code as a block
<%= %>   # executes the Ruby code and displays the result
```

Example

```
<ul>
<% @products.each do |p| %>
  <li><%= @p.name %></li>
<% end %>
</ul>
```

The output of anything in `<%= %>` tags is directly copied to the HTML output stream. To secure against HTML injection, use the `h()` function to `html_escape` the output.

Example

```
<%=h @user_entered_notes %>
```

Ruby on Rails – HTML Links

Following is the quick view on `link_to` and `mail_to`.

```
link_to "Name", :controller => 'post',
              :action => 'show',
              :id => @post.id

link_to "Delete", { :controller => "admin",
                   :action => "delete",
                   :id => @post },
        { :class => 'css-class',
          :id => 'css-id',
          :confirm => "Are you sure?" }

image_tag "spinner.png", :class => "image", :alt => "Spinner"

mail_to "info@invisible.ch", "send mail",
        :subject => "Support request by #{@user.name}",
        :cc => @user.email,
        :body => '....',
        :encoding => "javascript"

stylesheet_link_tag "scaffold", "admin", :media => "all"
```

Ruby on Rails – Session and Cookies

Sessions

To save data across multiple requests, you can use either the session or the flash hashes. A flash stores a value (normally text) until the next request, while a session stores data during the complete session.

```
session[:user] = @user
flash[:message] = "Data was saved successfully"
<%= link_to "login", :action => 'login' unless session[:user] %>
<% if flash[:message] %>
<div><%= h flash[:message] %></div>
<% end %>
```

It's possible to turn off session management.

```
session :off                    # turn session management off
session :off, :only => :action  # only for this :action
session :off, :except => :action # except for this action
session :only => :foo,          # only for :foo when doing HTTPS
    :session_secure => true
session :off, :only=>:foo, # off for foo,if uses as Web Service
    :if => Proc.new { |req| req.parameters[:ws] }
```

Cookies

Following is the syntax for setting cookies:

```
# Set a simple session cookie
cookies[:user_name] = "david"
# Set a cookie that expires in 1 hour
cookies[:login] = { :value => "XJ12", :expires => Time.now + 3600}
```

Following is the syntax for reading cookies:

```
cookies[:user_name] # => "david"
cookies.size         # => 2
```

Following is the syntax for deleting cookies:


```
cookies.delete :user_name
```

The option symbols for setting cookies are as follows:

- **value** - The cookie's value or list of values (as an array).
- **path** - The path for which this cookie applies. Defaults to the root of the application.
- **domain** - The domain for which this cookie applies.
- **expires** - The time at which this cookie expires, as a +Time+ object.
- **secure** - Whether this cookie is a secure cookie or not (default to false). Secure cookies are only transmitted to HTTPS servers.

Ruby on Rails – User Input Validations

Here is a list of validations that you can perform on a user input:

validates_presence_of

The following code checks that the last_name and the first_name contain data and are not NULL.

```
validates_presence_of :firstname, :lastname
```

validates_length_of

The following example shows various validations on a single file. These validations can be performed separately.

```
validates_length_of :password,
  :minimum => 8           # more than 8 characters
  :maximum => 16          # shorter than 16 characters
  :in => 8..16            # between 8 and 16 characters
  :too_short => 'way too short'
  :too_long => 'way to long'
```

validates_acceptance_of

The following code will accept only 'Y' value for an option field.

```
validates_acceptance_of :option
```

```
:accept => 'Y'
```

validates_confirmation_of

The fields password and password_confirmation must match and will be used as follows:

```
validates_confirmation_of :password
```

validates_uniqueness_of

The following code puts a condition for user_name to be unique.

```
validates_uniqueness_of :user_name
```

validates_format_of

The following code validates that a given email ID is in a valid format. It shows how you can use a regular expression to validate a field.

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

validates_numericality_of

It validates that a given field is numeric.

```
validates_numericality_of :value
                           :only_integer => true
                           :allow_nil => true
```

validates_inclusion_of

The following code checks that the passed value is an enumeration and falls in the given range.

```
validates_inclusion_of :gender,
                      :in => %w( m, f )
```

validates_exclusion_of

The following code checks that the given values do not fall in the given range.

```
validates_exclusion_of :age
                      :in => 13..19
```

validates_inclusion_of

The following code checks that the given values fall in the given range. This is the opposite of `validates_exclusion_of`.

```
validates_inclusion_of :age
                      :in => 13..19
```

validates_associated

It validates that the associated object is valid.

Options for all Validations

You can use the following options along with all the validations.

- **:message => 'my own errormessage'** Use this to print a custom error message in case of validation fails.
- **:on => :create or :update** This will be used in such cases where you want to perform validation only when record is being created or updated. If you use **:create**, then this validation works only when there is a create operation on database.

Ruby on Rails – Maths Functions

Consider a table object called **Person**. This table has fields like `age`, `first_name`, `last_name`, and `salary`.

The following code will return the average age of all the employees.

```
Person.average :age
```

The following code will return the maximum age of the employees.

```
Person.maximum :age
```

The following code will return the minimum age of the employees.

```
Person.minimum :age
```

The following code will return the sum of salaries of all the employees.

```
Person.sum :salary, :group => :last_name
```

The following code will count the number of records having age more than 26.

```
Person.count(:conditions => "age > 26")
```

The following code will count the total number of records.

```
Person.count
```

Ruby on Rails – Finders

Following are the ways to find out records with and without conditions:

The following code will find an author with ID 50.

```
Author.find(50)
```

The following code will find authors with ID 20, 30 and 40.

```
Author.find([20,30, 40])
```

The following code will find all the authors:

```
Author.find :all
```

The following code will find all the authors with first name *alam*.

```
Author.find :all  
          :condition => ["first_name =?", "alam" ]
```

The following code will find the first record of the authors with first name *alam*.

```
Author.find :first  
          :condition => ["first_name =?", "alam" ]
```

Options for Finders

You can use the following options along with the **find** function.

- **:order => 'name DESC'** Use this option to sort the result in ascending or descending order.
- **:offset => 20** Starts fetching records from offset 20.

- **:limit => 20** Returns only 20 records.
- **:group => 'name'** This is equivalent to SQL fragment GROUP BY.
- **:joins => LEFT JOIN ...'** Additional LEFT JOIN (rarely used).
- **:include => [:account, :friends]** This is LEFT OUTER JOIN with these model.
- **:select => [:name, :address]** Use this instead of SELECT * FROM.
- **:readonly => true** Use this to make objects write protected.

Dynamic Attribute-based Finders

You can use more dynamic functions to fetch values.

If there is a field **user_name**, then you can use the following to find records by username.

```
Person.find_by_user_name(user_name)
```

If there is a field **last_name**, then you can use the following to find records by last name.

```
Person.find_all_by_last_name(last_name)
```

If there are fields **user_name** and **password**, then you can use the following to find a record for a given username and password.

```
Person.find_by_user_name_and_password(user_name, password)
```

Ruby on Rails – Nested with-scope

The following example shows how we can nest **with_scope** to fetch different results based on requirements.

```
# SELECT * FROM employees
# WHERE (salary > 10000)
# LIMIT 10
# Will be written as
Employee.with_scope(
  :find => { :conditions => "salary > 10000",
    :limit => 10 }) do
  Employee.find(:all)
```

```
end
```

Now, check another example that shows how scope is cumulative.

```
# SELECT * FROM employees
# WHERE ( salary > 10000 )
# AND ( name = 'Jamis' ))
# LIMIT 10
# Will be written as
Employee.with_scope(
  :find => { :conditions => "salary > 10000",
    :limit => 10 }) do
  Employee.find(:all)
  Employee.with_scope(
    :find => { :conditions => "name = 'Jamis'" }) do
    Employee.find(:all)
  end
end
```

The following example shows how the previous scope is ignored.

```
# SELECT * FROM employees
# WHERE (name = 'Jamis')
# is written as
Employee.with_scope(
  :find => { :conditions => "salary > 10000",
    :limit => 10 }) do
  Employee.find(:all)
  Employee.with_scope(
    :find => { :conditions => "name = 'Jamis'" }) do
    Employee.find(:all)
  end
  # all previous scope is ignored
  Employee.with_exclusive_scope(
    :find => { :conditions => "name = 'Jamis'" }) do
```

```

        Employee.find(:all)
    end
end

```

Ruby on Rails – Callback Functions

During the life cycle of an active record object, you can hook into eight events:

- (-) save
- (-) valid?
- before_validation
- before_validation_on_create
- (-) validate
- (-) validate_on_create
- after_validation
- after_validation_on_create
- before_save
- before_create
- (-) create
- after_create
- after_save

Examples

```

class Subscription < ActiveRecord::Base
  before_create :record_signup
private
  def record_signup
    self.signed_up_on = Date.today
  end
end
class Firm < ActiveRecord::Base
  # Destroys the associated clients and
  # people when the firm is destroyed

```

```
before_destroy{
  |record|Person.destroy_all "firm_id= #{record.id}"
}
before_destroy{
  |record|Client.destroy_all "client_of= #{record.id}"
}
end
```