From Technologies to Solutions

# Building Dynamic Web 2.0 Websites with **Ruby on Rails**

Create database-driven dynamic websites with this open-source web application framework

A.P. Rajshekhar

# Building Dynamic Web 2.0 Websites with Ruby on Rails

Create database-driven dynamic websites with this open-source web application framework

**A.P. Rajshekhar**

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

# Building Dynamic Web 2.0 Websites with Ruby on Rails

First published: March 2008

Production Reference: 1200308

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847193-41-4

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

# Credits

**Author**
A.P. Rajshekhar

**Reviewer**
Walt Stoneburner

**Senior Acquisition Editor**
Douglas Paterson

**Development Editor**
Ved Prakash Jha

**Technical Editor**
Mithun Sehgal

**Editorial Team Leader**
Mithil Kulkarni

**Project Manager**
Abhijeet Deobhakta

**Project Coordinator**
Zenab Kapasi

**Indexers**
Hemangini Bari
Monica Ajmera

**Proofreader**
Angie Butcher

**Production Coordinator**
Shantanu Zagade

**Cover Work**
Shantanu Zagade

# About the Author

**A.P. Rajshekhar**, Senior Developer with Vectorform, has worked on enterprise-level web applications and game development. His endeavors include the development of a Learning Management System, Supply Management Solution, and Xbox-based games. He holds a Masters Degree in Computer Applications. He is a regular contributor to the Devshed Portal on topics ranging from server-side development (JEE/.Net/RoR) to mobile (Symbian-based development) and game development (SDL and OpenGL) with a total readership of more than 1.4 million.

Authoring a book is not an easy feat. However, the help and guidance from my family and friends helped me to author this book. First, I would like to thank Packt Publication for providing me an opportunity to work on such an exciting project. I would like to thank my parents for their constant encouragement. Special thanks to my friends Shrikant Khare and Sormita Chakraborty for their support and encouragement.

# About the Reviewer

**Walt Stoneburner** is a software architect with over 20 years of commercial application development and consulting experience. Fringe passions involve quality assurance, configuration management, and security. If cornered, he may actually admit to liking statistics and authoring documentation as well.

He's easily amused by programming language design, collaborative applications, and ASCII art. Self-described as a closet geek, Walt also evaluates software products and consumer electronics, draws cartoons, produces photography, writes humor pieces, performs slight of hand, enjoys game design, and can occasionally be found on ham radio.

Walt may be reached directly via email at `wls@wwco.com`. He publishes a tech and humor blog called the Walt-O-Matic at `http://www.wwco.com/~wls/blog/`. Rumors suggest that some of his strange videography may be found on iTunes.

Currently he is employed at Business & Engineering Systems Corporation as a lead engineer developing advanced software solutions for knowledge management.

Other book reviews and contributions include AntiPatterns and Patterns in Software Configuration Management (ISBN 0-471-32929-0, p. xi) and Exploiting Software: How to Break Code (ISBN 0-201-78695-8, p. xxxiii).

# Table of Contents

# Preface

Ruby on Rails is an open-source web application framework ideally suited to building business applications, accelerating and simplifying the creation of database-driven websites. It has been developed on the Ruby platform.

This book is a tutorial for creating a complete website with Ruby on Rails (RoR). It will teach you to develop database-backed web applications according to the Model-View-Controller pattern. It will take you on a joy ride right from installation to a complete dynamic website. All the applications discussed in this book will help you add exciting features to your website. This book will show you how to assemble RoR's features and leverage its power to design, develop, and deploy a fully featured website.

## What This Book Covers

*Chapter 1* gives you an overview of the features of Ruby and RoR, as well as providing the various ways of installing, configuring, and testing both Ruby and RoR.

*Chapter 2* introduces you to the basics of Ruby as well as the main concepts and components of RoR.

*Chapter 3* makes you understand the design of tables according to the conventions of RoR, creation of scaffolds for tables, and changing the scaffolds according to the requirements.

*Chapter 4* gives you details about how to set up the User Management module for the website called TaleWiki.

*Chapter 5* makes you familiar with the Login Management and Comment Management modules for TaleWiki.

*Chapter 6* introduces you to the Migrations and Layouts involved in setting up the template for TaleWiki.

*Chapter 7* describes the tagging functionality being implemented for the enhanced search usability.

*Chapter 8* provides you with the implementation of AJAX for TaleWiki.

*Chapter* 9 deals with the development of an interface for the administration.

*Chapter 10* gives you the steps for deploying the website.

## What You Need for This Book

- Operating System: Windows 2000 or above / Redhat Fedora core 1.0 or above
- Database: MySQL 4.9 or above
- Editor: Notepad/Vim or Emacs
- Browser: Firefox 1.5 or above/ Internet Explorer 6.0 or above

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "For example, to add instance attributes named `author` and `genre` to the `Tale` class, you will do it as follows:"

A block of code will be set as follows:

```
class Tale
   @author
   @genre
   @tale_body
end
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
class Tale
   @author
   @genre
   @tale_body
end
```

Any command-line input and output is written as follows:

```
c:\InstantRails\rails_apps\ > rails talewiki
```

**New terms** and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "In the next page, without entering any data, click on the **Create** button."

Important notes appear in a box like this.

Tips and tricks appear like this.

## Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

## Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the Example Code for the Book

Visit `http://www.packtpub.com/files/code/3414_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

## Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **Submit Errata** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

## Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

# 1 Getting Started with Ruby and RoR

'Which is the best framework for web application development?' This question is asked in different ways and forms. The answer, however, always remains the same—'The one that enhances productivity'. The next obvious query would be 'In that case which is the framework that enhances productivity?' After some debates and deliberations, we can conclude, 'A framework that reduces Boilerplate code and also reduces the learning curve is the one that increases productivity.'

If you look around, there are an abundance of frameworks that cater to web application development. But most of them fail in one of the two points that govern productivity. Either the framework reduces the Boilerplate code or it is easier to learn. Achieving a balance between the two is seen as a tough task. It is here that Ruby and Ruby on Rails (or RoR as it is fondly called), score above most of their contemporaries. How they achieve this balance is what we will be looking at in this book.

We will be developing a website throughout the book, each chapter adding something new to the website. This chapter will lay the groundwork of introducing you to Ruby and RoR. It will also tell you the ways to install and configure Ruby and RoR—one-click as well as manual installation, and finishing with techniques to test your installation.

## Ruby and RoR—The Next Level in Dynamic Web Development

It is always a good idea to know about the specifications of the tool with which one has to work, before handling the tool. In our context, the tools are Ruby and RoR—Ruby as the language and RoR as the framework built upon Ruby.

## Ruby

In 1995, Yukihiro Matsumoto released the first version of Ruby and this added one more language to the ever-growing toolkit of application developers. The current stable version is 1.8.6. According to the TIOBE Programming Community Index, it is the fastest growing language. So, what makes it the fastest growing one among the languages? To understand this, let us first understand the reason behind the creation of Ruby. The main reason given by Mr. Matsumoto for creating Ruby was that he wanted a scripting language that would optimize the way a programmer would develop the software. This means that the features of Ruby are such that they optimize the way the software is developed. What are these features? Let us have a look at them:

- **Interpreted**: Ruby is an interpreted language. Therefore, whenever you make a change to the source code, you need not compile the code and then run it to see the effect of the change. As a result of this feature, the code-compile-run cycle becomes the code-run cycle.
- **Purely Object-Oriented**: Ruby is purely object-oriented. That means that everything in Ruby is an object which includes primitive data-types and numbers. In addition, it supports all the features required by an **Object-Oriented Language**.
- **Functional**: Ruby supports functional programming using blocks.
- **Duck Typing**: It is also known as Dynamic Typing. Ruby decides about the type of variable while the program is running by looking at the value contained in the variable at that instant. In other words, if an object looks like a duck, sounds like a duck, then it is a duck!
- **Automatic Memory Management**: You would know it as Garbage Collection. As in any Very High-Level Language (VHLL), Ruby provides Garbage Collection out-of-the-box, thus you need not worry about physical memory leaks.
- **Threading**: The current stable version of Ruby provides 'almost' platform independent threading using green threads (threads used at the user-space level are known as green threads.) I said 'almost' because Ruby threads are simulated in the VM rather than running as native OS threads.
- **Reflection**: Ruby provides a program with the ability to 'look at itself' while running. This ability is known by different terms, such as reflection, introspection, and so on. Using reflection, a program can modify certain aspects of itself during execution, or create a completely new object at runtime based on the requirements at that time.

Looking at the features we just discussed, you could definitely see that the creator's reason holds true. The way imperative programming features have been balanced with functional programming is the proof of that. It is on such a foundation that RoR has been built.

## Ruby on Rails (RoR)

RoR is a recent entrant in the world of web application frameworks. So how come such a new player on the block not only stands on its own but can also challenge veteran players of the likes of J2EE/JEE? The answer does not just lie in the functionalities. The other aspect that governs the acceptance of a framework is its philosophy. Hence, we will have to look at both the aspects of RoR—functionality as well as philosophy. Keep in mind that the philosophy holds true for all the versions of RoR.

## Philosophy

The philosophy of RoR depends on two principles:

- **DRY**: Don't Repeat Yourself or the DRY principle, if applied properly, reduces the duplication of tasks within a project. Duplication of any kind, within a project, leads to difficulty in modification and maintenance and inconsistency. In RoR, you can see this principle at work in almost everything—from the reusable components in the form of plug-ins to the way database tables are mapped.
- **Convention-over-Configuration** (**CoC**): Configuration has taken over the web application frameworks so much that even a simple task such as applying 'compulsory field' validation for just one field requires entries in an XML file. In RoR, the principle is to supply information about only those aspects that are different from usual application settings. The ORM (Object Relational Mapping) framework provided by RoR is an example of the Convention-over-Configuration principle. Unless you specify a different name for an ORM object, the object uses the name of the table to which it is mapped. Whereas in the case of configuration-based ORM frameworks, such as Hibernate, the mapping of each table along with its columns has to be given in the configuration file. So, a change in the schema means a change in the configuration file. However, in the case of RoR, a change in the schema doesn't mean a change in the object unless the name of the table itself changes. We will see more about the ORM framework in Chapter 2.

## Features

The features based on the philosophy of DRY and Convention-over-Configuration principles are what make RoR so attractive for the development of dynamic websites. The features that showed the way for alternative methods for implementation of various server-side techniques are:

- **Automatic setup of Application structure**: If you have worked with J2EE technologies, this would come as a pleasant surprise. The structure of any application need not be created manually. Just one command and the complete structure including folders and basic web files such as `index.html` will be generated for you. Therefore, no more hunting for third party tools such as those that provide the initial setup or setting up the structure manually.
- **Generation of Boilerplate Code**: Every application has certain blocks of code that are essentially the same for all other applications of the same type or category. Such blocks are called Boilerplate code. One of the examples of Boilerplate code is the code block setting up a connection to the database. The same code can be used with different applications with only a little change. Though this is the case, most of the frameworks do not provide any inbuilt mechanism to reduce this 're-invention of the wheel'. RoR avoids the duplication using scaffolding. In essence, a scaffold contains the bare minimum of code to accomplish tasks such as connecting to the database, setting up a log, and so on. Scaffolds reflect the DRY principle that RoR adheres to.
- **Dynamic mapping of classes to database schemas**: No web application can go online without having a database as its back-end. ORM frameworks have eased the database access. However, the configuration aspect reduces any advantage to the developer. In the case of RoR, ORM does not need any configuration. At runtime, RoR reads and maps the schema based on the names of classes and corresponding tables using reflection and meta-programming. Moreover, what the developer gets is more productivity.
- **Ajax at the core of presentation**: Ajax is the technology that provides interactivity to websites without becoming intrusive. All the current server-side technologies claim to support Ajax, but the support is peripheral and not at the core. You would have to download new libraries, configure them, and then start the develop-compile-deploy-test cycle again. Whereas in RoR, Ajax is part of the core libraries. So when you install RoR, Ajax support is also made available to you. Using them is as easy as when you use any other library provided by RoR.

- **Batteries included**: RoR contains many more libraries that provide for essentially all the requirements of a dynamic website including layout management, mailing, and so on. If you look at these libraries, you will understand that they are, in fact, fully-fledged components in themselves, representing different services provided by a website or a portal.

That completes the roundup of features of the 'tools' that we are going to use to build our website. The next step is to install and configure our 'tools' so that we can get started with our task.

## Installing and Configuring Ruby and RoR

RoR can be installed in two ways:

1. Manual installation after installing Ruby
2. One-click installer that installs Ruby and RoR, which includes Apache web-server and MySQL database server

If you already have Apache and MySQL installed, then manual installation is the better way as it installs only RoR.

### Manual Installation

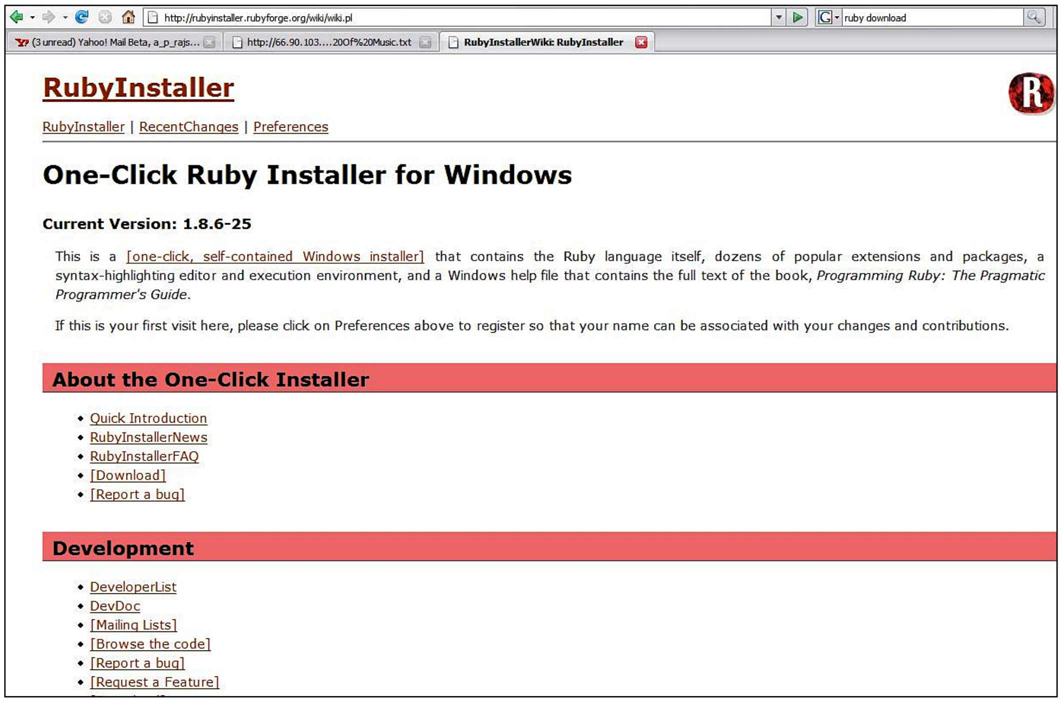
There are three main steps for manually installing RoR, which are:

- Downloading and installing Ruby
- Updating gem
- Installing RoR using gem

In this case, the RoR installation is done over the internet. So from the second step itself, ensure that the internet is connected throughout the installation.

## Downloading and Installing Ruby

First, grab the Ruby installer for windows at `http://rubyinstaller.rubyforge.org/wiki/wiki.pl`.



http://rubyinstaller.rubyforge.org/wiki/wiki.pl

(3 unread) Yahoo! Mail Beta, a_p_rajs... | http://66.90.103....20Of%20Music.txt | RubyInstallerWiki: RubyInstaller

RubyInstaller

RubyInstaller | RecentChanges | Preferences

One-Click Ruby Installer for Windows

Current Version: 1.8.6-25

This is a [one-click, self-contained Windows installer] that contains the Ruby language itself, dozens of popular extensions and packages, a syntax-highlighting editor and execution environment, and a Windows help file that contains the full text of the book, *Programming Ruby: The Pragmatic Programmer's Guide*.

If this is your first visit here, please click on Preferences above to register so that your name can be associated with your changes and contributions.

About the One-Click Installer

- Quick Introduction
- RubyInstallerNews
- RubyInstallerFAQ
- [Download]
- [Report a bug]

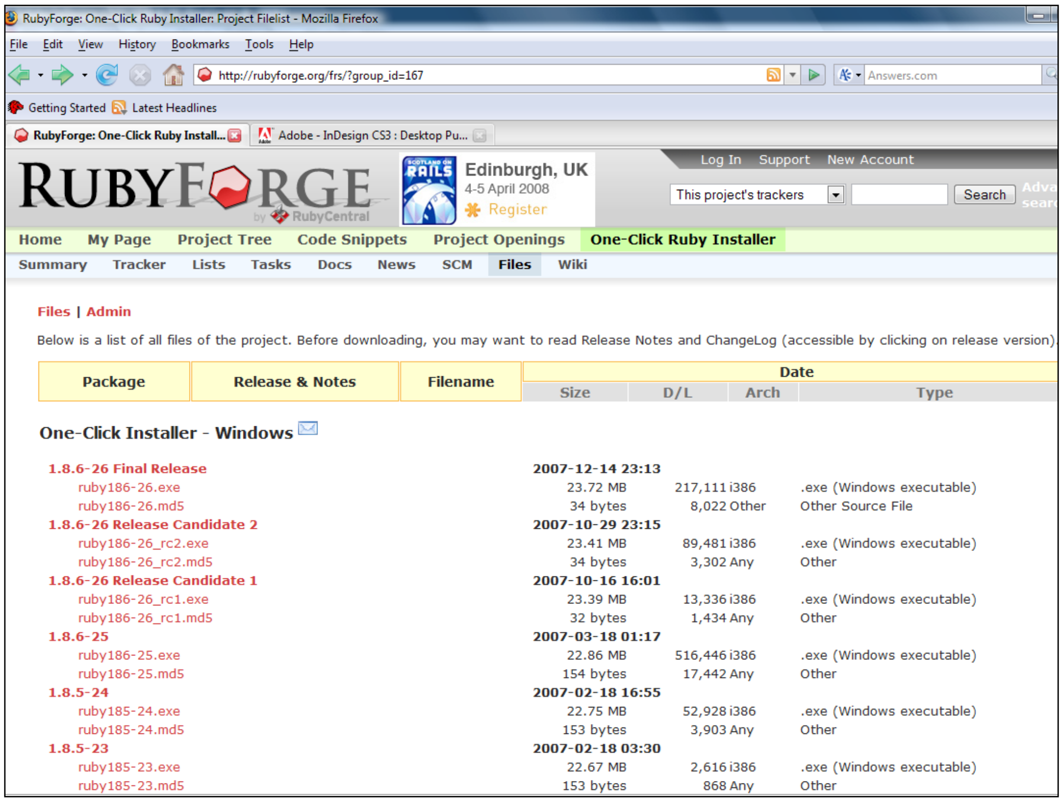
Development

- DeveloperList
- DevDoc
- [Mailing Lists]
- [Browse the code]
- [Report a bug]
- [Request a Feature]

The previous figure shows the main page for the one-click Ruby installer (do not confuse it with one-click RoR installer). It provides a list of links that provide details about the one-click installer. Here are the steps for downloading and installing Ruby:

- From the list, select **[Download]** link.

- On clicking the **[Download]** link, you will be taken to the page listing the downloadable release version of the installers.



RubyForge: One-Click Ruby Installer: Project Filelist - Mozilla Firefox

http://rubyforge.org/frs/?group_id=167

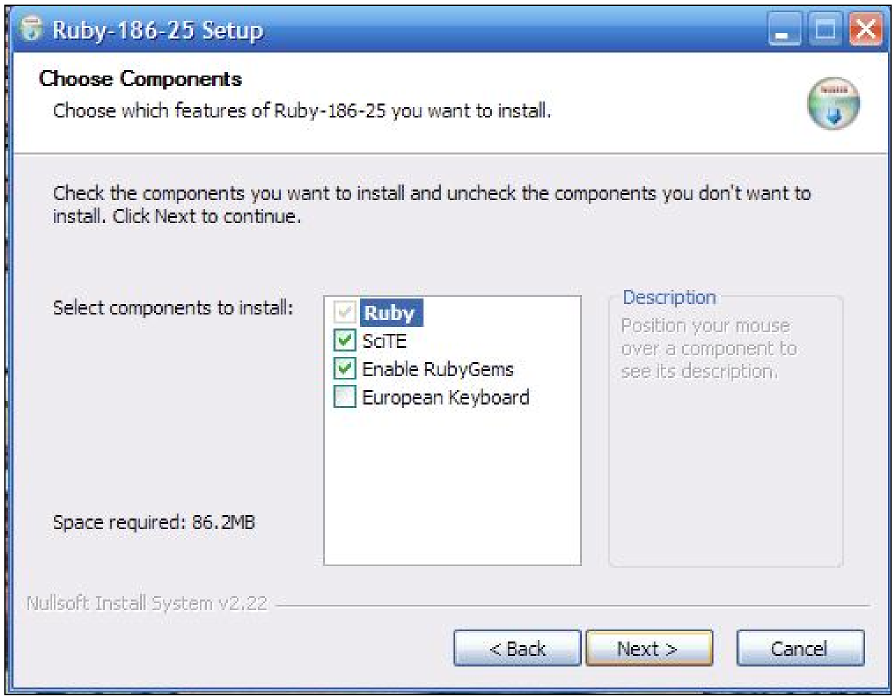
Files | Admin

Below is a list of all files of the project. Before downloading, you may want to read Release Notes and ChangeLog (accessible by clicking on release version).

| Package | Release & Notes | Filename | Date / Size | D/L | Arch | Type |
|---|---|---|---|---|---|---|
| One-Click Installer - Windows | | | | | | |
| | 1.8.6-26 Final Release | | 2007-12-14 23:13 | | | |
| | | ruby186-26.exe | 23.72 MB | 217,111 | i386 | .exe (Windows executable) |
| | | ruby186-26.md5 | 34 bytes | 8,022 | Other | Other Source File |
| | 1.8.6-26 Release Candidate 2 | | 2007-10-29 23:15 | | | |
| | | ruby186-26_rc2.exe | 23.41 MB | 89,481 | i386 | .exe (Windows executable) |
| | | ruby186-26_rc2.md5 | 34 bytes | 3,302 | Any | Other |
| | 1.8.6-26 Release Candidate 1 | | 2007-10-16 16:01 | | | |
| | | ruby186-26_rc1.exe | 23.39 MB | 13,336 | i386 | .exe (Windows executable) |
| | | ruby186-26_rc1.md5 | 32 bytes | 1,434 | Any | Other |
| | 1.8.6-25 | | 2007-03-18 01:17 | | | |
| | | ruby186-25.exe | 22.86 MB | 516,446 | i386 | .exe (Windows executable) |
| | | ruby186-25.md5 | 154 bytes | 17,442 | Any | Other |
| | 1.8.5-24 | | 2007-02-18 16:55 | | | |
| | | ruby185-24.exe | 22.75 MB | 52,928 | i386 | .exe (Windows executable) |
| | | ruby185-24.md5 | 153 bytes | 3,903 | Any | Other |
| | 1.8.5-23 | | 2007-02-18 03:30 | | | |
| | | ruby185-23.exe | 22.67 MB | 2,616 | i386 | .exe (Windows executable) |
| | | ruby185-23.md5 | 153 bytes | 868 | Any | Other |

- From the given list, select the `.exe` link for the latest release and save the file in your preferred location. In our case the version to be downloaded is **ruby186-25.exe**.
- Double click on the file to be installed to start the installation process.
- The first screen that will be presented to you should be the **License Agreement**. Read the license carefully and click on **I Agree**.

The next screen will present you with the components to be installed.



**Choose Components**

Choose which features of Ruby-186-25 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] Ruby
- [✓] SciTE
- [✓] Enable RubyGems
- [ ] European Keyboard

Description

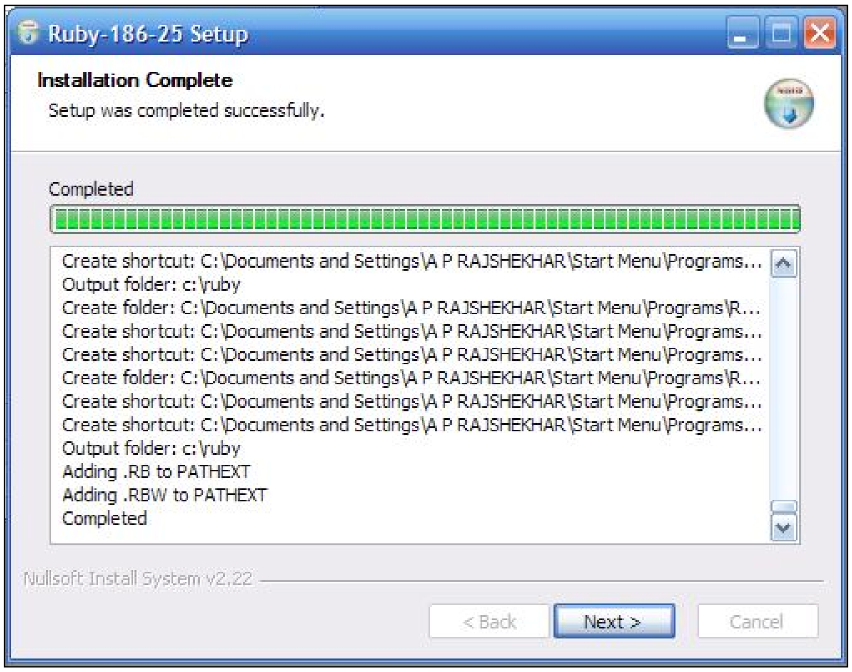
Position your mouse over a component to see its description.

Space required: 86.2MB

Nullsoft Install System v2.22

- Keep the default choices and click **Next**. Of these **SciTE** is a programmer's editor and **RubyGems** is Ruby's package manager and updater.
- Next, choose where Ruby should be installed. It's always advisable to install in the root of any drive instead of a sub-folder. Ruby commands may not work correctly if the sub-folder is deeply nested. For example, if you want to install it in drive C:, then give `c:\ruby` as the value for **Destination Folder**. Also keep in mind not to specify any folder name with spaces in it, as it may create problems while installing RoR.
- Next, provide the name for the Start Menu entry for Ruby installation. Keep the default name and click **Install**.

If the installation completes without any problem, then you will see the following screen:



```
Create shortcut: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs...
Output folder: c:\ruby
Create folder: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs\R...
Create shortcut: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs...
Create shortcut: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs...
Create folder: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs\R...
Create shortcut: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs...
Create shortcut: C:\Documents and Settings\A P RAJSHEKHAR\Start Menu\Programs...
Output folder: c:\ruby
Adding .RB to PATHEXT
Adding .RBW to PATHEXT
Completed
```

- Click on the **Next >** button to complete the installation process.
- The last screen presented by the installer should give you an option to view the Readme file. If you wish to read it, check the **Show Readme** checkbox and then click on **Finish** to complete the installation.

That completes the Ruby installation. The next step is to update the installation using gem.

## Updating Gem

Gem is the name of the utility supplied with Ruby in order to manage, install, and update the Ruby installation in an easy way. The second step in the manual installation of RoR is updating the Ruby installation so that if a new package or an update for any of the package is available, then the complete installation can be made up-to-date.

Before we begin, if you are behind a proxy, open the command prompt and give the following command:

```
>Set HTTP_PROXY=http://<proxy_address>:<proxy_port>
```

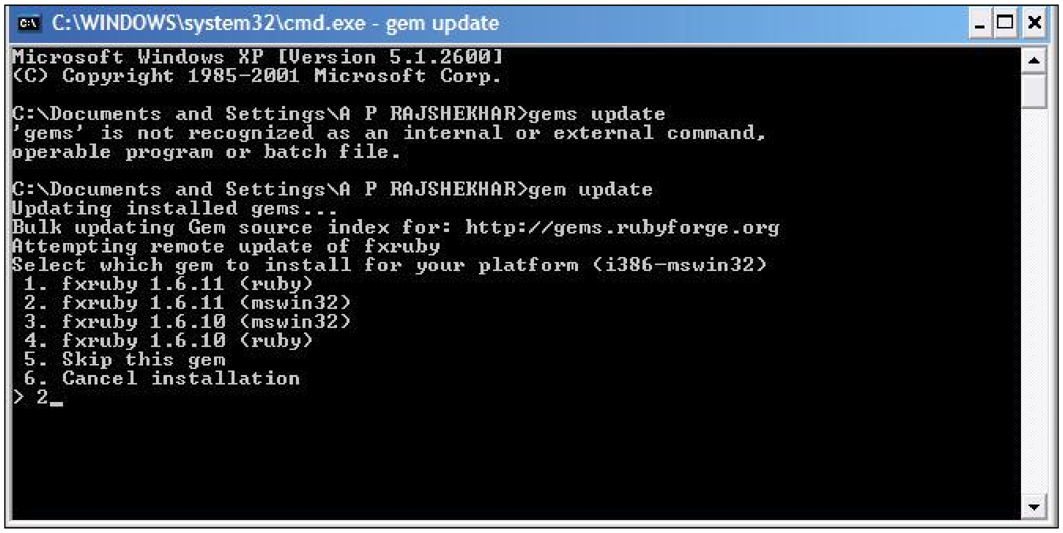
For example, if the address of the proxy server is 192.168.1.1 and the port number is 9090, then you would have to give the following at the command prompt:

```
>set HTTP_PROXY=http://192.168.1.1:9090
```

Next, give the following command at the prompt:

```
>gem update
```

You will get the following reply after the last step:



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\A P RAJSHEKHAR>gems update
'gems' is not recognized as an internal or external command,
operable program or batch file.

C:\Documents and Settings\A P RAJSHEKHAR>gem update
Updating installed gems...
Bulk updating Gem source index for: http://gems.rubyforge.org
Attempting remote update of fxruby
Select which gem to install for your platform (i386-mswin32)
 1. fxruby 1.6.11 (ruby)
 2. fxruby 1.6.11 (mswin32)
 3. fxruby 1.6.10 (mswin32)
 4. fxruby 1.6.10 (ruby)
 5. Skip this gem
 6. Cancel installation
> 2_
```

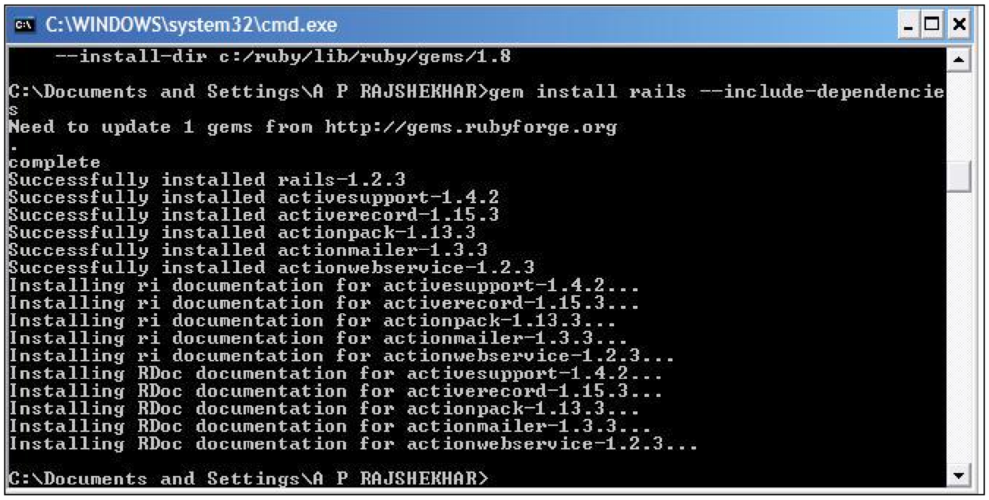
Select the compatible version for your platform. Here I choose option 2 which is the latest for Windows. The difference between <ruby> and <mswin32> is that the former is a pure Ruby-based package and the latter is the packages natively compiled for Windows. If there is more than one package to be updated, then more 'choice menus' will be presented to you. The point to remember is to choose the number corresponding to the latest version of the package natively compiled for Windows.

That completes the update gem step. Next, let us install RoR.

## Installing RoR

This is the last and the easiest part of the installation process. Just one command and RoR shall be installed. At the prompt, issue the following command:

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



```
    --install-dir c:/ruby/lib/ruby/gems/1.8

C:\Documents and Settings\A P RAJSHEKHAR>gem install rails --include-dependencie
s
Need to update 1 gems from http://gems.rubyforge.org
.
complete
Successfully installed rails-1.2.3
Successfully installed activesupport-1.4.2
Successfully installed activerecord-1.15.3
Successfully installed actionpack-1.13.3
Successfully installed actionmailer-1.3.3
Successfully installed actionwebservice-1.2.3
Installing ri documentation for activesupport-1.4.2...
Installing ri documentation for activerecord-1.15.3...
Installing ri documentation for actionpack-1.13.3...
Installing ri documentation for actionmailer-1.3.3...
Installing ri documentation for actionwebservice-1.2.3...
Installing RDoc documentation for activesupport-1.4.2...
Installing RDoc documentation for activerecord-1.15.3...
Installing RDoc documentation for actionpack-1.13.3...
Installing RDoc documentation for actionmailer-1.3.3...
Installing RDoc documentation for actionwebservice-1.2.3...

C:\Documents and Settings\A P RAJSHEKHAR>
```

The command should give the messages as shown in the previous figure. Congrats! RoR is now installed on your system.

For those working on GNU/Linux, only the first step would differ. To install Ruby, grab the latest tar file from `http://www.ruby-lang.org/en/`. Then give the following command at the prompt:

```
tar -zxvf <ruby_tar_file>
```

Then go into the directory created by the tar command. Inside the directory issue following commands:

```
./configure; make; make install
```

That's it. Ruby is ready to be explored.

## One-Click RoR Installation

The one-click installer is, in fact, a zip package containing everything that you need. All you need to do is download it and unzip it to a directory of your choice. Even though, it is in a single package, the installation needs to be configured. Therefore, in essence, there are two steps:

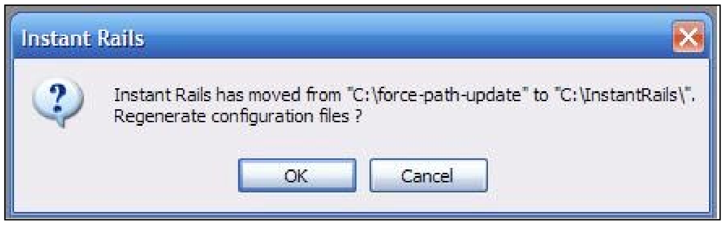
## Download and Unzip the Instant Rails

- First, go to the following address:

  `http://instantrails.rubyforge.org/wiki/wiki.pl`.
- From the links listed on the page click on the **[Download]** link.
- On the next page, select the package corresponding to the latest release from the list.
- Once the download has completed, unzip the package into the directory of your choice. I use the `InstantRails` directory.

## Configure Instant Rails Installation

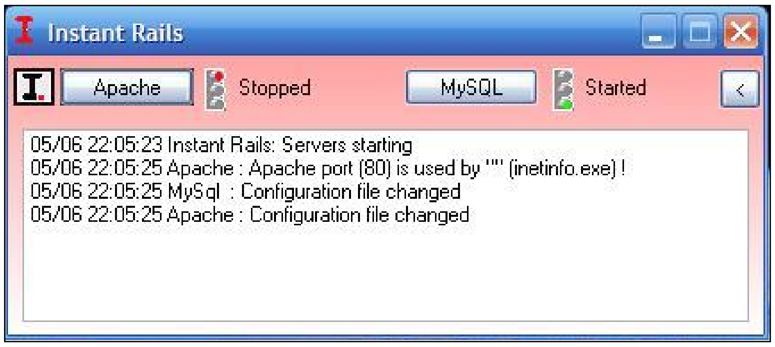
The first step is to configure the environment of the installation. To do so, click on the `InstantRails.exe` file within the directory of Instant Rails. You will be presented with the following dialog box:



Instant Rails has moved from "C:\force-path-update" to "C:\InstantRails\". Regenerate configuration files ?

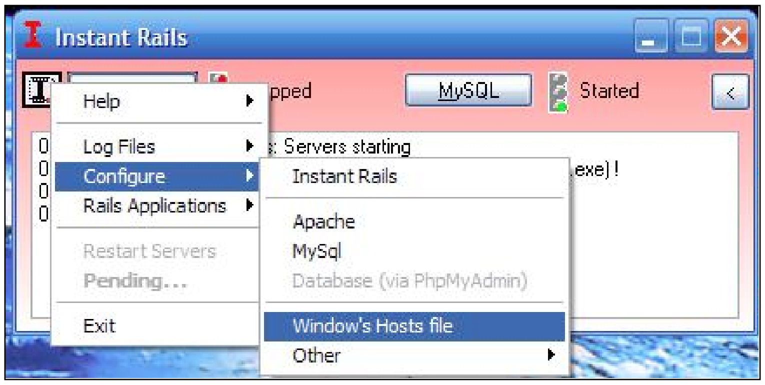
When the dialog box just shown appears, click **OK.** It will configure the environment variables for the Instant Rails directory.

Once configuration is done, you will be presented with the main application window. What the configuration does is that it updates the configuration files for the Apache web-server and the MySQL database server. It also starts these servers.



```
05/06 22:05:23 Instant Rails: Servers starting
05/06 22:05:25 Apache : Apache port (80) is used by "" (inetinfo.exe) !
05/06 22:05:25 MySql  : Configuration file changed
05/06 22:05:25 Apache : Configuration file changed
```

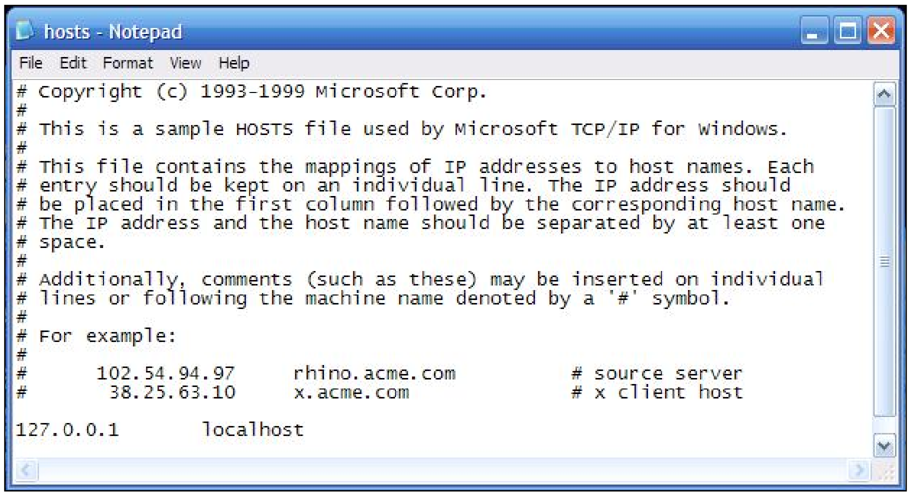
Next, we have to tell Windows about how to find and launch our application. To achieve this we have to change the **Window's Hosts File**. Click on the button labelled **I** and choose **Configure | Window's Hosts file**.



The host file will be opened in NOTEPAD. The file should contain the following line:

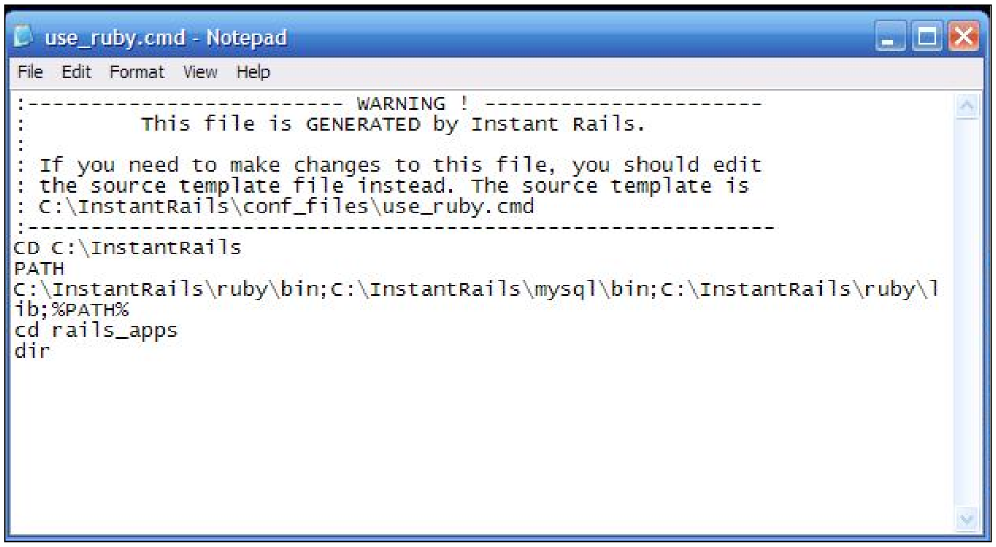
```
127.0.0.1          localhost
```

If it is not there, it has to be added manually. **Save** the changes and exit Notepad.



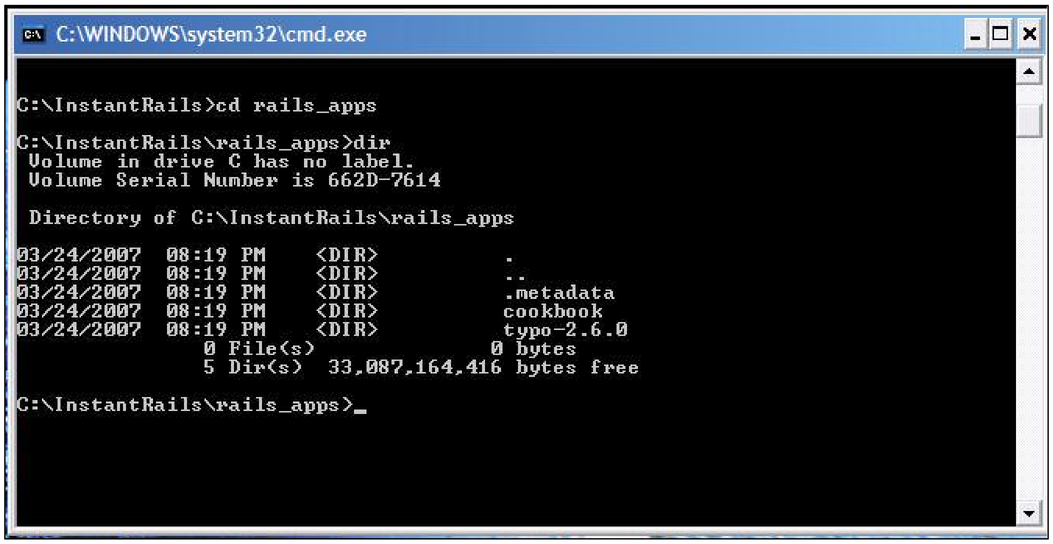
```
# Copyright (c) 1993-1999 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

127.0.0.1       localhost
```

Now, we have to set the path to the Ruby directory of the Instant Rails installation. To do so, open the `use_ruby.cmd` file within the Instant Rails installation directory. Then, add the **<Instant_Rails_directory>\ruby\lib**; line to the **PATH** entry in the file, where `<Instant_Rails_directory>` is the path of the directory where Instant Rails have been installed. Now, **Save** the file to `%WINDIR%\system32` folder. In the case of Windows 2000, `%WINDIR%` refers to `WINNT` folder, and in the case of Windows XP, it is the `Windows` folder. By doing this, giving the command `use_ruby` enables you to use Instant Rails without changing anything in the installation directory. After the addition, the content of the file will be as follows:



```
:------------------------- WARNING ! ----------------------
:         This file is GENERATED by Instant Rails.
:
: If you need to make changes to this file, you should edit
: the source template file instead. The source template is
: C:\InstantRails\conf_files\use_ruby.cmd
:-------------------------------------------------------------
CD C:\InstantRails
PATH
C:\InstantRails\ruby\bin;C:\InstantRails\mysql\bin;C:\InstantRails\ruby\l
ib;%PATH%
cd rails_apps
dir
```

On giving the `use_ruby` command, if you get a screen similar to the following, the configuration has been successful. That completes the one-click RoR installation.



```
C:\InstantRails>cd rails_apps

C:\InstantRails\rails_apps>dir
 Volume in drive C has no label.
 Volume Serial Number is 662D-7614

 Directory of C:\InstantRails\rails_apps

03/24/2007  08:19 PM    <DIR>          .
03/24/2007  08:19 PM    <DIR>          ..
03/24/2007  08:19 PM    <DIR>          .metadata
03/24/2007  08:19 PM    <DIR>          cookbook
03/24/2007  08:19 PM    <DIR>          typo-2.6.0
               0 File(s)              0 bytes
               5 Dir(s)  33,087,164,416 bytes free

C:\InstantRails\rails_apps>_
```

## Testing the Installation

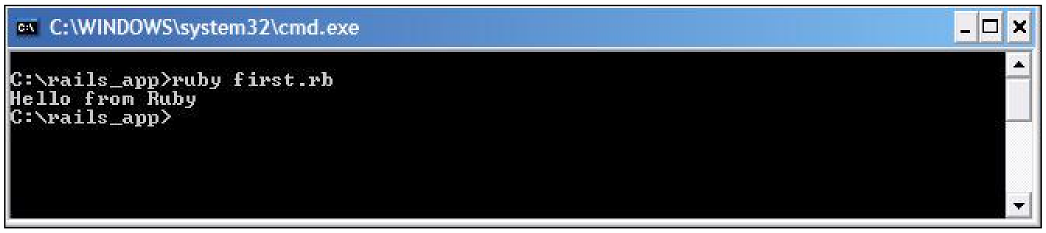
The installation is successful as much as the process is concerned. However, it is always a good idea to test the installation. From this point onwards, I will be using the rails_apps directory as the base directory for the RoR application that will be developed within this book. The installation has to be tested for two components:

### Ruby

Fire up the editor of your choice (I will be using ScITE) and enter the following code: `print 'Hello Ruby'`. Save it in a file named first.rb, and place the file in the rails_apps directory. Then drop into the command prompt and change into the rails_app directory (if you are using Instant Rails, then the rails_app directory would be inside the Instant Rails directory). Then run the file with the following command:

```
>ruby first.rb
```

The result should be as shown below.



```
C:\rails_app>ruby first.rb
Hello from Ruby
C:\rails_app>
```

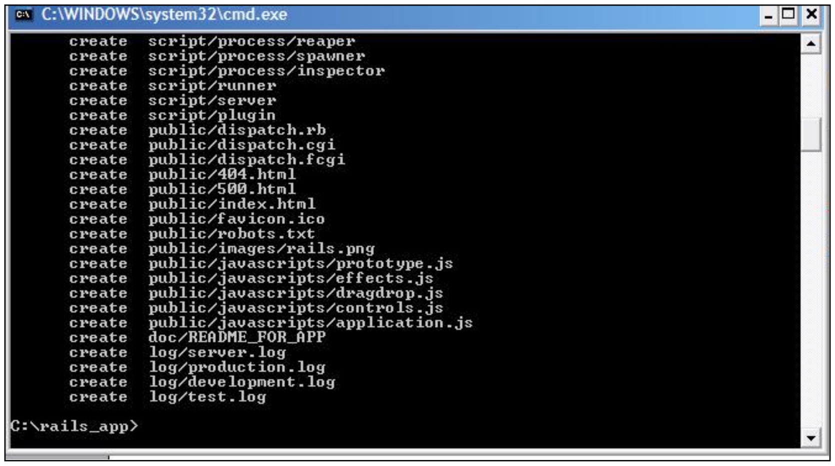
Anything apart from the output shown means you will have to check the installation and configuration of Ruby. This test is more important in the case of an Instant Rails installation. The reason is that in the case of any manual installation, if Ruby does not work, then the RoR installation would not be successful. However, in the case of Instant Rails, everything comes as a bundle. If it succeeds, then Ruby supplied with the bundle is working fine. That completes the first part of 'Testing the Installation'.

### RoR

Inside the rails_apps directory, issue the following command:

```
>rails test_app
```

If you get the screen shown next, then your RoR installation is OK. What has happened is that RoR has generated the whole file structure for the application. Even certain files that work as placeholders have been generated by just one command. Impressive, isn't it?



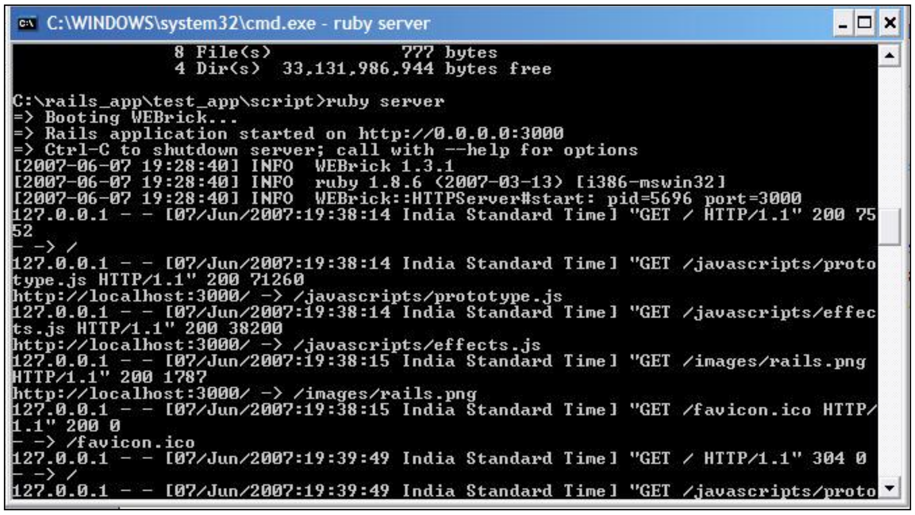
C:\WINDOWS\system32\cmd.exe

```
      create  script/process/reaper
      create  script/process/spawner
      create  script/process/inspector
      create  script/runner
      create  script/server
      create  script/plugin
      create  public/dispatch.rb
      create  public/dispatch.cgi
      create  public/dispatch.fcgi
      create  public/404.html
      create  public/500.html
      create  public/index.html
      create  public/favicon.ico
      create  public/robots.txt
      create  public/images/rails.png
      create  public/javascripts/prototype.js
      create  public/javascripts/effects.js
      create  public/javascripts/dragdrop.js
      create  public/javascripts/controls.js
      create  public/javascripts/application.js
      create  doc/README_FOR_APP
      create  log/server.log
      create  log/production.log
      create  log/development.log
      create  log/test.log

C:\rails_app>
```

The next step is to check the server provided by RoR. Give the following command after changing into the `test_app` folder.

```
> ruby script/server
```

As a response to the command, you should see the messages shown in the following screen saying that it is booting up the WebRick server.



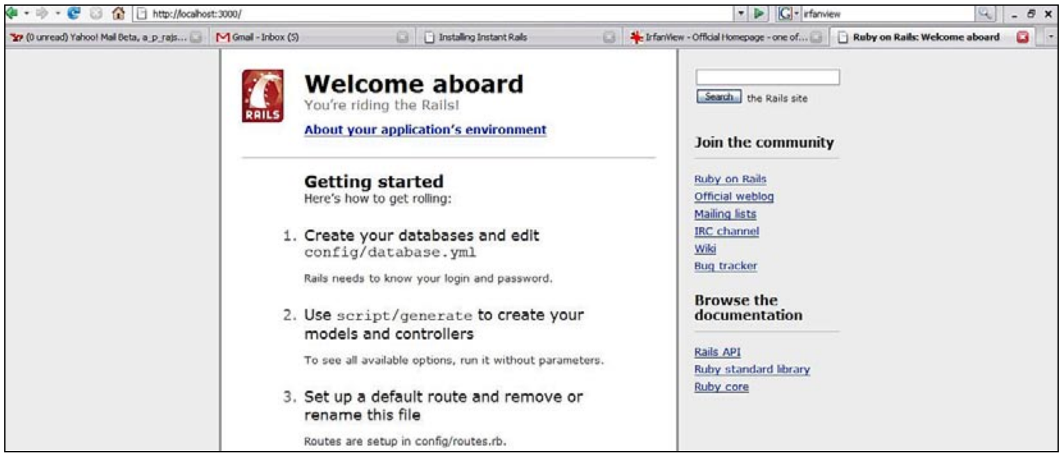
C:\WINDOWS\system32\cmd.exe - ruby server

```
               8 File(s)            777 bytes
               4 Dir(s)  33,131,986,944 bytes free

C:\rails_app\test_app\script>ruby server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2007-06-07 19:28:40] INFO  WEBrick 1.3.1
[2007-06-07 19:28:40] INFO  ruby 1.8.6 (2007-03-13) [i386-mswin32]
[2007-06-07 19:28:40] INFO  WEBrick::HTTPServer#start: pid=5696 port=3000
127.0.0.1 - - [07/Jun/2007:19:38:14 India Standard Time] "GET / HTTP/1.1" 200 75
52
- -> /
127.0.0.1 - - [07/Jun/2007:19:38:14 India Standard Time] "GET /javascripts/proto
type.js HTTP/1.1" 200 71260
http://localhost:3000/ -> /javascripts/prototype.js
127.0.0.1 - - [07/Jun/2007:19:38:14 India Standard Time] "GET /javascripts/effec
ts.js HTTP/1.1" 200 38200
http://localhost:3000/ -> /javascripts/effects.js
127.0.0.1 - - [07/Jun/2007:19:38:15 India Standard Time] "GET /images/rails.png
HTTP/1.1" 200 1787
http://localhost:3000/ -> /images/rails.png
127.0.0.1 - - [07/Jun/2007:19:38:15 India Standard Time] "GET /favicon.ico HTTP/
1.1" 200 0
- -> /favicon.ico
127.0.0.1 - - [07/Jun/2007:19:39:49 India Standard Time] "GET / HTTP/1.1" 304 0
- -> /
127.0.0.1 - - [07/Jun/2007:19:39:49 India Standard Time] "GET /javascripts/proto
```

WebRick is a project embeddable server provided by RoR that resides in the `script` folder of the application which in this case is `test_app`.

Next, open the browser of your choice and provide the following URL: `http://localhost:3000`.

You will be presented with the following screen. If you get anything else, then it means that you need to go through the steps for installation and configuration once again. That covers testing the manual installation of RoR. So what about Instant Rails? That is what is coming up next.


http://localhost:3000/
(0 unread) Yahoo! Mail Beta, a_p_rajs...
Gmail - Inbox (5)
Installing Instant Rails
IrfanView - Official Homepage - one of...
Ruby on Rails: Welcome aboard
irfanview
RAILS
Search
the Rails site

**Welcome aboard**
You're riding the Rails!
About your application's environment

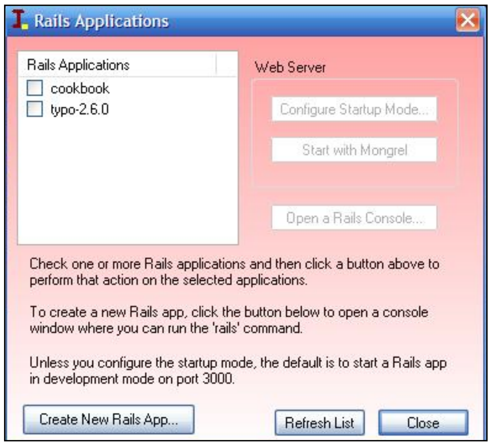
**Getting started**
Here's how to get rolling:

1. Create your databases and edit `config/database.yml`
   Rails needs to know your login and password.
2. Use `script/generate` to create your models and controllers
   To see all available options, run it without parameters.
3. Set up a default route and remove or rename this file
   Routes are setup in config/routes.rb.

**Join the community**
Ruby on Rails
Official weblog
Mailing lists
IRC channel
Wiki
Bug tracker

**Browse the documentation**
Rails API
Ruby standard library
Ruby core

To test Instant Rails's RoR installation, first stop the Apache server by selecting **Apache | stop**. This step is necessary so that the Rails server is provided by Instant Rails. Next, select **I | Rails Applications | Manage Rails Applications...**. It will popup the following window:

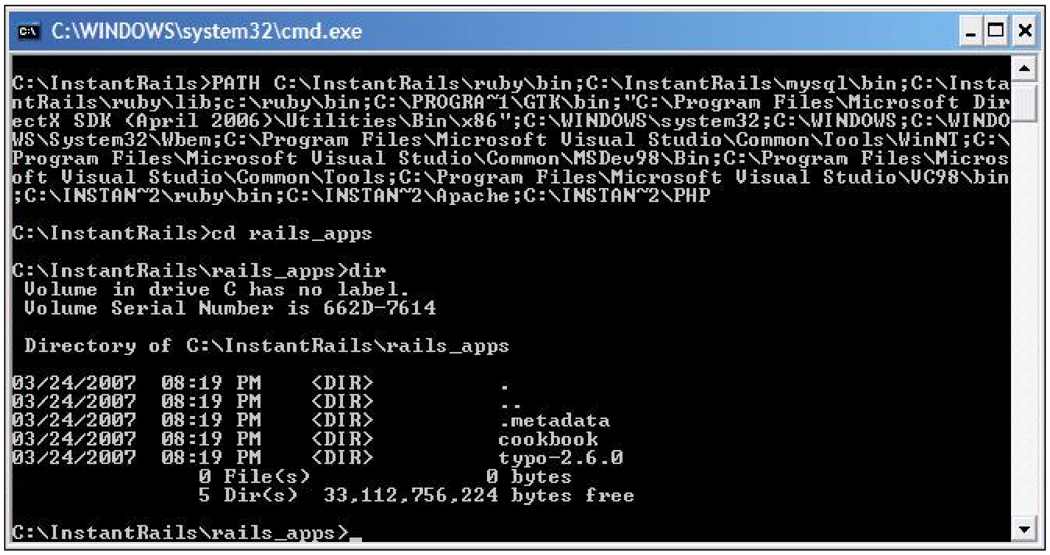


Check one or more Rails applications and then click a button above to perform that action on the selected applications.

To create a new Rails app, click the button below to open a console window where you can run the 'rails' command.

Unless you configure the startup mode, the default is to start a Rails app in development mode on port 3000.

Click on the **Create New Rails App...** button. It will drop you into the shell at the `rails_app` directory of the Instant Rails install folder as shown in the following screen:

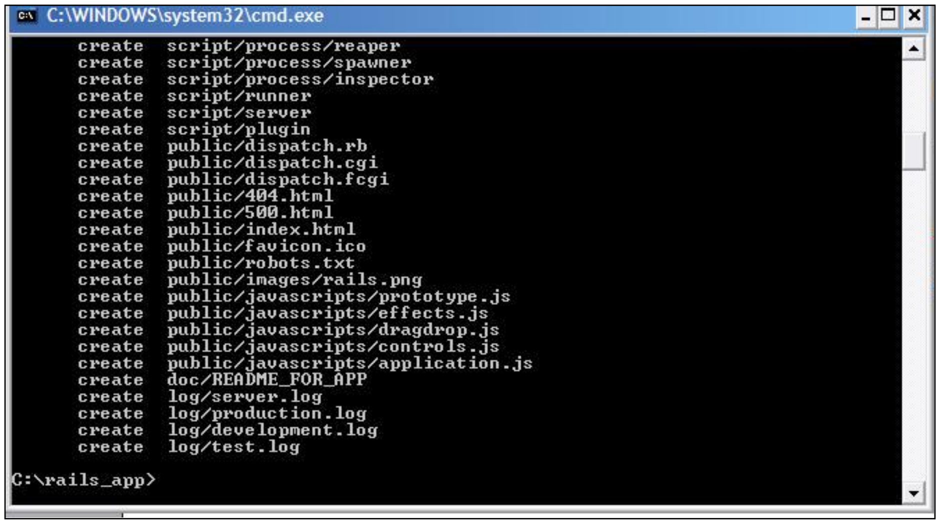


```
C:\InstantRails>PATH C:\InstantRails\ruby\bin;C:\InstantRails\mysql\bin;C:\Insta
ntRails\ruby\lib;c:\ruby\bin;C:\PROGRA~1\GTK\bin;"C:\Program Files\Microsoft Dir
ectX SDK (April 2006)\Utilities\Bin\x86";C:\WINDOWS\system32;C:\WINDOWS;C:\WINDO
WS\System32\Wbem;C:\Program Files\Microsoft Visual Studio\Common\Tools\WinNT;C:\
Program Files\Microsoft Visual Studio\Common\MSDev98\Bin;C:\Program Files\Micros
oft Visual Studio\Common\Tools;C:\Program Files\Microsoft Visual Studio\VC98\bin
;C:\INSTAN~2\ruby\bin;C:\INSTAN~2\Apache;C:\INSTAN~2\PHP

C:\InstantRails>cd rails_apps

C:\InstantRails\rails_apps>dir
 Volume in drive C has no label.
 Volume Serial Number is 662D-7614

 Directory of C:\InstantRails\rails_apps

03/24/2007  08:19 PM    <DIR>          .
03/24/2007  08:19 PM    <DIR>          ..
03/24/2007  08:19 PM    <DIR>          .metadata
03/24/2007  08:19 PM    <DIR>          cookbook
03/24/2007  08:19 PM    <DIR>          typo-2.6.0
               0 File(s)              0 bytes
               5 Dir(s)  33,112,756,224 bytes free

C:\InstantRails\rails_apps>_
```

Then give the rails command as follows:

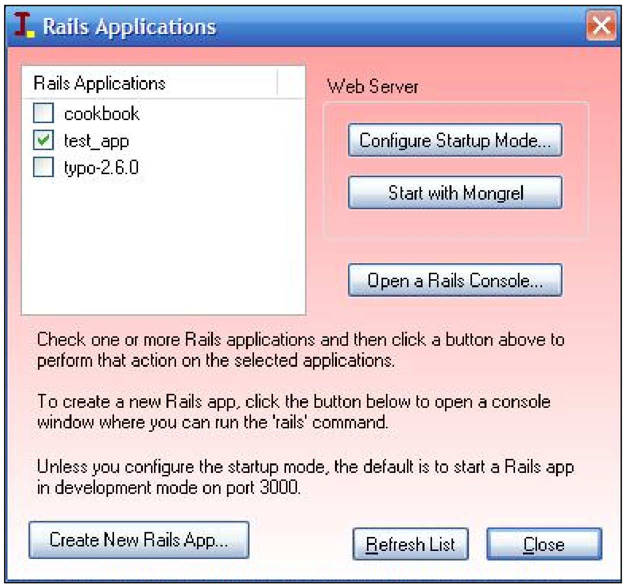
```
>rails test_app
```

If the screen you get is the same as shown it means the Instant Rails installation is correct. It is the same as that of the one which you saw for the manual installation.



```
      create  script/process/reaper
      create  script/process/spawner
      create  script/process/inspector
      create  script/runner
      create  script/server
      create  script/plugin
      create  public/dispatch.rb
      create  public/dispatch.cgi
      create  public/dispatch.fcgi
      create  public/404.html
      create  public/500.html
      create  public/index.html
      create  public/favicon.ico
      create  public/robots.txt
      create  public/images/rails.png
      create  public/javascripts/prototype.js
      create  public/javascripts/effects.js
      create  public/javascripts/dragdrop.js
      create  public/javascripts/controls.js
      create  public/javascripts/application.js
      create  doc/README_FOR_APP
      create  log/server.log
      create  log/production.log
      create  log/development.log
      create  log/test.log

C:\rails_app>
```

Now, select **I | Rails Applications | Manage Rails Applications...**, and in the pop-up window, select the check box corresponding to **test_app**. Then click **Start with Mongrel**. The popup window will appear as the following screen:



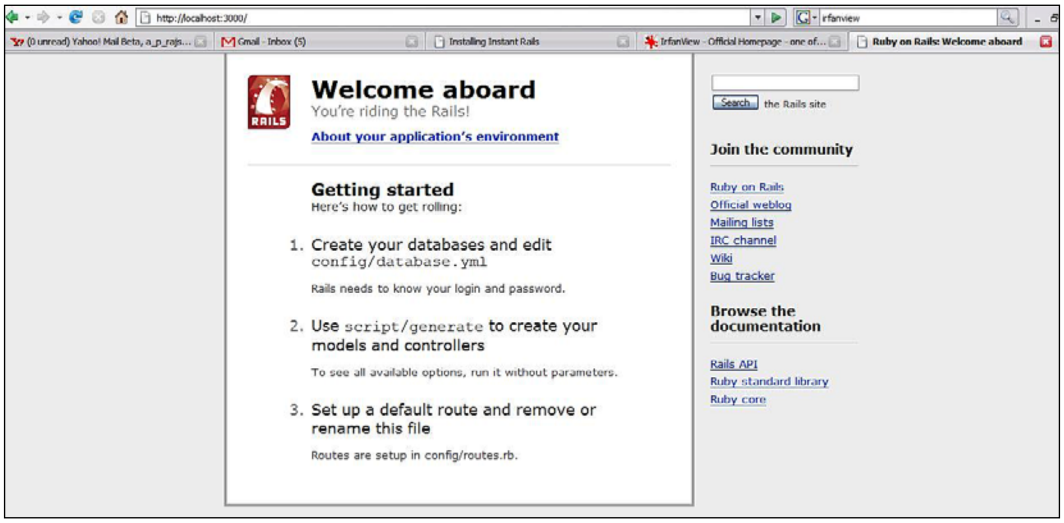
- [ ] cookbook
- [x] test_app
- [ ] typo-2.6.0

Check one or more Rails applications and then click a button above to perform that action on the selected applications.

To create a new Rails app, click the button below to open a console window where you can run the 'rails' command.

Unless you configure the startup mode, the default is to start a Rails app in development mode on port 3000.

The Mongrel server will be started as a command window. Next, in the browser give the URL as `http://localhost:3000` to bring up the default index page of the `test_app` application. If it looks like the following screen that means you are set to go into the exciting world of RoR.



Welcome aboard

You're riding the Rails!

About your application's environment

Getting started

Here's how to get rolling:

1. Create your databases and edit `config/database.yml`

   Rails needs to know your login and password.

2. Use `script/generate` to create your models and controllers

   To see all available options, run it without parameters.

3. Set up a default route and remove or rename this file

   Routes are setup in config/routes.rb.

This completes the testing phase of the installation. From here on, I will be using Instant Rails as the development environment.

## Summary

That brings us to the end of the first chapter. In this chapter, you have had an overview of the features of Ruby and RoR. It also took you through the various ways of installing, configuring, and testing of both Ruby and RoR.

The next chapter will take you deeper into Ruby and RoR, as it will deal with the components and concepts of both Ruby and RoR. Sit tight as this is just the beginning...

# 2
# Getting to Know Ruby and RoR

In the last chapter, the focus was on the specifications and installation of the 'tools', if I continue using the analogy of 'tools.' By the same analogy—until the user understands which control provides what functionality—the tool cannot be used to its maximum potential. So it is necessary to understand which library provides what functionality and which component maps to what specification for each and every tool. Ruby and RoR is no exception to this.

RoR builds upon the functionalities provided by Ruby. Thus, by understanding how Ruby works, you can know about the building blocks of RoR. That in turn, will help you to have a clearer picture of how the different components of RoR fit into the bigger picture. The chapter will first introduce you to the basic concepts of Ruby. Then it will move on to the basic concepts and components of RoR. Finally, the chapter will be completed with an example of RoR, which can be considered as 'Hello World' in RoR.

## Ruby—the Basics

To understand Ruby, you will have to understand the concepts that are fundamental to Ruby. These concepts are:

- Classes
- Inheritance
- Module
- Data Types
- Blocks and Iterators
- Exception Handling
- Data Structures

Of these, the first two are Object-Oriented concepts. Let us have a look at each of these concepts and the way Ruby implements them. However, you will have to keep one point in mind. The discussion in this section is not 'the definitive guide' to Ruby. The focus of this section is to provide you with the fundamentals of Ruby so that you can understand RoR better.

## Classes, Attributes, Methods, and Objects

Classes, attributes, methods, and objects are the core of any Object-Oriented language. How they are implemented and how they can be used, differs from language to language. How they are implemented in Ruby?—that's what I am going to discuss now.

### Classes

A class is a blueprint that represents a section of the real world objects. For example, a class 'Tale' would represent a real world tale (or a story). A class is an abstract representation of a real world object, including its characteristics and functionalities. Hence, it doesn't occupy space in memory during the execution of the program that contains the class.

There are two types of classes—*close-ended* and *open-ended*. If a class is *close-ended*, then new functionalities cannot be added to it without inheriting the class. A C++ or a Java class is *close-ended* because you cannot add a new functionality to it without inheriting or subclassing it.

On the other hand, if a class is *open-ended*, then new functionalities can be added to it without inheriting it. One of the important aspects of the Ruby class is that it is *open-ended*. It means you can add new functionalities at any point of time.

In Ruby, the class declaration and definition happens at the same time. A class is declared using a **class** keyword. The definition goes between **class** <*class_name*> and **end**. For example, to declare and define a class named `Tale,` you have to write:

```
class Tale
end
```

The class declared and defined just now is an empty class as it doesn't contain any attributes or methods. If you compare the class definition with say, that of Java or C++, you will observe that there is no parenthesis demarcating the body of the class. Also, Ruby doesn't rely on indentation for the demarcation of blocks. The only thing that you have to keep in mind is to provide an **end** statement at the end of each block—whether the block is an attribute or a class.

The name of a class should always begin with a capital letter.

## Attributes

Like all other Object-Oriented languages, Ruby too has the provision for defining the fields or attributes for a class. The attributes are the variables that describe the qualities of a class. To continue with our 'Tale' example, a real world 'Tale' will have an author, a genre, and so on. So, the name of the author and genre are the qualities that describe a 'Tale.' Now if we compare a 'Tale' to a class, then author and genre will become its *attributes*.

A class can have two kinds of attributes—instance attributes and class-level attributes. Instance attributes describe the qualities of an instance of a class, whereas the class-level attributes describe the qualities of all the current instances of a class. In other words, the instance attributes are bound to a specific instance of the class but the class-level attributes are bound to the class itself. So the value of a class-level attribute will be same across all the instances of that class and the value of an instance attribute will differ with each instance. The static variable is like a global variable stored in a class that can be accessed by all the instances of that class. The change done to the static variable by one of the instances will be seen by the other instances. Another term for an instance attribute is instance variable and for class-level attribute is class-level variable.

In Ruby, you must define the instance attributes using the `@` symbol. To define the class-level attributes you will have to use the `@@` symbol. For example, to add instance attributes named `author` and `genre` to the `Tale` class, you will do it as follows:

```
class Tale
   @author
   @genre
   @tale_body
end
```

## Methods

Methods define the functionality provided by a class. Simply stated, methods tell about what a class can do. You can define three types of methods—one that neither accepts any parameter nor returns any value, one that accepts parameters but doesn't return any value, and lastly, one that accepts parameters as well as returns values. Let's say that every 'Tale' can *tell* the way through which that 'Tale' is told to others, then 'Tale' will have a method 'tell'.

You can add a method to the class using the `def` keyword. The end of the method is denoted by the `end` keyword. For example, you would add a method `tell` to the class `Tale` in the following way:

```
class Tale
   @author
   @genre
   @tale_body
#method to display the tale
   def tell
      print @tale_body
   end
end
```

A method can accept values through parameters. Parameters are declared as variables in parenthesis. If there is only one parameter then parenthesis is not required. If the `tell` method takes only *part of the tale* to be told then the method would be like:

```
class Tale
   @author
   @genre
   @tale_body
   #method to display part of the tale
    def tell part_to_be_told
        start_index=@tale_body.index(part_to_be_told)
           print @tale_body\
              [start_index+part_to_be_told.length,@tale_body.length]
   end
end
```

A method can return a value by using the `return` keyword. If the `Tale` class contains a method that returns the name of the `author`, it would be:

```
class Tale
   @author
   @genre
   @tale_body
   #method to display part of the tale
```

```
    def tell part_to_be_told
       start start_index=@tale_body.index(part_to_be_told)
          print @tale_body\
             [start_index+part_to_be_told.length,@tale_body.length]
    end
    def get_author
       return @author
    end
 end
```

There is a special type of method that is used to initialize the instance attributes of a class. Such a method is known as **constructor**. The specialty of the constructor is that it is called when an instance of that class is created. Also, it doesn't return any value. In Ruby, the constructor is denoted by the `initialize` method. Let's add a constructor to our `Tale` class. The instance attributes are moved to the constructor.

```
class Tale
   def initialize(author, genre, body)
      @author=author
      @genre=genre
      @tale_body=body
   end
   def tell part_to_be_told
      start_index=@tale_body.index(part_to_be_told)
         print @tale_body\
            [start_index+part_to_be_told.length,@tale_body.length]
   end
   def get_author
      return @author
   end
end
```

Another special kind of method is related to attributes. Attributes can be read and written using *getters and setters*. A `Getter` is a method that has the same name as that of the attribute whose value it is to get. To make it clearer I will rewrite `get_author` to make it a `getter` as follows:

```
class Tale
   def initialize(author, genre, body)
      @author=author
      @genre=genre
      @tale_body=body
   end
   def tell part_to_be_told
      start_index=@tale_body.index(part_to_be_told)
         print @tale_body\
            [start_index+part_to_be_told.length,@tale_body.length]
   end
   #getter for author attribute
      def author
      @author
   end
end
```

Next is the `setter`. A `setter` is a method having the same name as that of the attribute followed by the variable containing the value to be set. The variable containing the value is kept in parenthesis and is assigned to the attribute name. A `setter` for the `author` attribute of our `Tale` class will be:

```
   class Tale
      def initialize(author, genre, body)
         @author=author
         @genre=genre
         @tale_body=body
   end
      #method to display part of the tale
      def tell part_to_be_told
         start_index=@tale_body.index(part_to_be_told)
            print @tale_body\
               [start_index+part_to_be_told.length,@tale_body.length]
      end
      #getter for author attribute
         def author
         @author
      end
      #setter for author attribute
         def author=(newAuthor)
         @author=newAuthor
         end
   end
```

## Objects

Until now, I have been using the term 'instance of a class'. What does this actually mean? As already said, a class doesn't occupy any memory. Memory is only allocated when an instance of the class is created. The instance of a class is another term for an object of the class. If the class is a blueprint then an object is the implementation of the blueprint. For example, `tale_of_mermaid` will be the object of the `Tale` class.

A new Ruby object is created by calling a `new` method on the class. The `new` method is an implicit method. It means that each class will be supplied with the new method by the Ruby Interpreter. You don't have to define the `new` method on your own. It automatically calls the `initialize` method. Therefore, the following code will create an object of the `Tale` class named `mermaid_tale`

```
mermaid_tale=Tale.new("unknown","fairy_tale", "once upon a
time…. ")
```

Once you create the instance of a class, you can call its methods. If you want to tell only the first part of the Cinderella tale you will call:

```
mermaid_tale.tell("first")
```

Similarly, attributes can be get and set by calling getters and setters on the object of the `Tale` class. So, if you want to set the name of the `author` to `Jim Henson`, then you will call the setter as:

```
mermaid_tale.author= "Jim Henson"
```

And to get the name of the `author` you can call the getter on the object. The `#` sign is used to replace the value of the variable that comes after it. In the following statement, `mermaid_tale.author` will be replaced by its value as it comes after `#`.

```
print "Author is #mermaid_tale.author"
```

## Inheritance

In general, inheritance is something that is transferred from parents to children. The inheritance can be of any type. The physical attributes of parents and the mental faculties of parents are some of the qualities in our parents that we have inherited. In the same way, a class can inherit methods and attributes of another class. In an Object-Oriented approach, this is known as Inheritance. Inheritance is also known as specialization.

Let's say you want to have a class that deals specifically with *fantasy tales*. Let's name the new class `FantasyTale`. Apart from the attributes specified in the `Tale` class, the `FantasyTale` class will have the type of creature that the `Tale` focuses on. Following is the definition of the `FantasyTale` class:

```
class FantasyTale < Tale
   def initialize (author, genre, body, creature)
      super (author, genre, body)
      @creature = creature
end
end
```

The `< Tale` tells Ruby that the `FantasyTale` class inherits from the `Tale` class. Hence, `FantasyTale` will have all the methods that we have defined in the `Tale` class. Here, the `Tale` class is the super class and `FantasyTale` is the subclass. To pass values to the super class, the keyword `super` needs to be used. When you create an object of a subclass, the object can access the methods of the super class. The following code creates an instance of `FantasyTale`. Then it calls the `tell` method of `Tale` (which is FantasyTale's super class) to display part of the tale.

```
fantasy = FantasyTale.new("Jim Henson", "Fantasy", "Once upon a
time….", "mermaid")
fantasy.tell ("Once")
```

Ruby supports multiple inheritance through mixins. However, it is out of the scope of this chapter to discuss mixins.

## Modules

Modules are the collection of classes or methods. The main use of modules is to prevent namespace clashes that may occur when you try to use different classes with the same name. Modules are defined by the `module` and `end` keywords. For example, `Tale` and `FantasyTale` classes can be encapsulated in a module as follows:

```
module Tales
   class Tale
   end
   class FantasyTale < Tale
   end
end
```

To call a module, you will need to use the `::` operator. For example, to call the `Tale` class you will need to use the following statement:

```
T1 = Tales :: Tale.new("Jim Henson", "Fantasy", "Once upon a time….")
```

Next, let us look at data types.

# Data Types

In Ruby, the data type of a variable is determined at runtime. However, this doesn't mean that there is no primitive data type support in Ruby. Ruby has the following primitive data types.

## Number

Number represents an integer. It can be any integer—positive or negative. Places within the numbers can be demarcated using an underscore. For example:

```
tale_age=1000
or
tale_age=1_000
```

Both the above forms are valid.

## Float

Any decimal number is a float data type. A variable of a float type can have a number with decimal points as well as scientific notation. Hence you can define a float variable in either of the following ways:

```
diameter=3.145
or
diameter=3.14e-05
```

## String

Any alphanumeric character surrounded by quotes is considered a string. Both single as well as double quotes denote a string. Both the following codes are valid strings:

```
s= "It's a beautiful life"
or
s1= 'It's a beautiful life'
```

If you observe, when the alphanumeric sequence contains a single quoted character or characters, they need to be enclosed in double quotes. The string can be accessed as a list as you have seen in the `tell` method.

There is a special kind of string called Symbol. It can be considered as a lightweight string. It is mostly used when the string need not be shown on the screen. A Symbol is like a common variable except that it is prefixed with a colon ( : ). For example, both the following are valid symbols:

```
:part_of_story
or
:next_part
```

One point to keep in mind is that data types in Ruby are objects. The data types being discussed next are objects.

## Blocks and Iterators

Ruby provides unnamed blocks to group a set of statements together. It also provides a technique known as an iterator to go through a collection. Let's see how they can be used to make development simpler.

### Blocks

Code blocks are of two kinds—*named* and *unnamed* (or *anonymous*). Classes and methods come under *named* blocks. *Anonymous* blocks are denoted with curly braces ({ }). They can also start with the `do` keyword and end with the `end` keyword. You can use either curly braces ({ }) or `do`/`end`. Loops also come under *anonymous* blocks. An example of *anonymous* block can be opening a file. In code it will be:

```
5.times{
     print "Hello \n"
}
```

Blocks can have arguments too. In other words, they can work like a slimmed down version of methods. Throughout this chapter, I will be using methods and functions interchangeably. The block arguments are surrounded by pipe characters and separated by comma. So to pass the message to be printed inside the block, you can rewrite the above code as:

```
5.times{|msg|
   print msg
}
```

Use of blocks will become clearer when we discuss iterators, and that is coming up next.

### Iterators

Iterators do what their name suggests—they iterate over a collection. Iteration can be done using the `each` keyword. The way in which iterator is used is:

```
<collection>.each {|<var>| <operation on var>}
```

Here, `<collection>` can be an array or a list of any kind and `var` is the variable. For example, let's say that array `alist` has been defined as follows:

```
alist=[1,2,3.4,"foo"]
```

Then by using iterators you can print the elements of the array in the following way:

```
alist.each {|item| puts item}
```

It will give you the following result:

```
>ruby iterator_example.rb
1
2
3.4
foo
>Exit code: 0
```

So how do blocks and iterators help in more common contexts such as reading and writing to a file? As an answer to this question let us come back to the `Tale` class. Everything is fine and dandy until now. You can create different tale objects, get and set author's name, and display a part of the tale. However, once the object is destroyed, the values of that object are lost. In other words, the values present in individual instance attributes are not saved. This is where reading and writing of a file (such as file I/O) comes into the picture. Using blocks and iterators, file I/O can become really easy. Let's add a method that writes the values to a file.

```
class Tale
   #constructor
   def initialize(author, genre, body)
      @author=author
      @genre=genre
      @tale_body=body
   end
   #method to display part of the tale
   def tell part_to_be_told
      start_index=@tale_body.index(part_to_be_told)
         print @tale_body\
            [start_index+part_to_be_told.length,@tale_body.length]
   end
   #getter for author attribute
   def author
      @author
   end
   #setter for author attribute
   def author=(newAuthor)
      @author=newAuthor
   end
   def write_to_file
   #opens a file and does the writing within it
   File.open ('tale.txt','w') { |file|
       file.puts @author
            file.puts @genre
            file.puts @tale_body
            }
   end
end
```

Here everything is done within the block. Now let's add a `read_from_file` method that will read a file and print the contents onto the screen.

```
class Tale
   #constructor
   def initialize(author, genre, body)
      @author=author
      @genre=genre
      @tale_body=body
   end
   #method to display part of the tale
   def tell part_to_be_told
      start_index=@tale_body.index(part_to_be_told)
         print @tale_body\
            [start_index+part_to_be_told.length,@tale_body.length]
   end
   #getter for author attribute
   def author
      @author
   end
   #setter for author attribute
   def author=(newAuthor)
      @author=newAuthor
   end
   #method to write the attribute values to a file
   def write_to_file
   #opens a file and does the writing within it
   File.open ('tale.txt','w') { |file|
       file.puts @author
       file.puts @genre
       file.puts @tale_body
       }
   end
   #method to read from a file
   def read_from_file
   File.readlines('tale.txt').each { |line|
       puts line
       }
   end
end
```

Here, the file is read and each line is iterated over using each, which is then passed into the block by passing it as the argument. Then the value of the argument is printed. That's how iterators and blocks work together.

## Exception Handling

Exceptions are error conditions that interrupt the normal execution of a program. Exceptions can occur due to many reasons including I/O errors and trying to divide by zero. It is always a good practice to handle exceptions. To handle exceptions, Ruby provides the `raise` and `rescue` clauses. Every block containing a logic that can give raise exceptions is kept inside a `begin`/`end` block as shown below

```
begin
   #logic
   rescue
   #handle error condition
end
```

For example, to catch all the exceptions that can occur within a block, you will have to write:

```
begin
   #logic
   rescue Exception
   #handle error condition
end
```

Let's say that in the `read_from_file` method of the `Tale` class, while opening the file, an exception can occur. The exception can occur due to many reasons, such as the file does not exist, the path to the file is wrong or the user does not have permission to access it. So, let's add an error handling block. For that you will have to rewrite the `read_from_file` method as follows:

```
def write_to_file
#opens a file and does the writing within it
   begin
   File.open ('tale.txt','w') { |file|
       file.puts @author
       file.puts @genre
       file.puts @tale_body
       }
   rescue Exception => ex
   $stderr.print "File open failed "
   end
end
```

In this case, it catches any kind of exception. That is how exceptions are handled in Ruby.

## Data Structures

Most of the power and ease that Ruby provides to the developers comes from its inbuilt data structures. The most commonly used data structures are:

- Arrays
- Hashes

Of these the former is index-based and the latter is key-based.

### Arrays

An array is a list that holds a collection of items. The `Array` class of Ruby provides the array related functionalities. The main difference between an array in Ruby and an array in a language—such as Java—is that an array in Ruby is a dynamic data structure. A list of books is an example where an array can be used. An array containing a list of books can be created as follows:

```
books = ["The Treasure Island", "Don Quixote"]
```

The `books` array can be accessed using the index or a number starting at `0`. For example, using the following statement, you can access the first element of the array, `The Treasure Island`.

```
books[0]
```

You have to keep in mind that arrays in Ruby are zero indexed. An interesting relationship between arrays and strings is that strings can be accessed as arrays. If you remember the definition of the `tell` method, we have accessed the body variable as an array. Next, let us look at hashes.

### Hashes

Arrays are useful as long as you do not want to associate the values with any other type other than the numeric index. Let us say that you would like to use the name of the author as the index instead of a number. In such a case, you will need to use a hash instead of an array. Similar to array, hash is also a class. To define a `hash` as having the details of a book including `title`, `author`, and `genre`, the statement will be:

```
book = {"title" => "The Treasure Island"
          ,  "author" => "R.L.Stevenson"
             "genre" => "Adventure"
       }
```

To access the title of the book, you will do as follows:

```
book["title"]
```

The last statement will give `The Treasure Island` as output. That completes the fast track introduction to the basic concepts of Ruby. Next, let us see how RoR builds itself on Ruby.

[ The # is an interesting character. When used outside single or double quotes, the line following it is treated as a comment. However, inside single or double quotes, it is treated as a value replacement. ]

## RoR—Concepts and Components

Now that the basics of Ruby have been introduced, let us move on to the next stage—RoR. If you ask the question, 'What is RoR?', the most common answer will be, 'RoR is a Ruby-based framework that implements the MVC pattern'. There are two key points in this answer:

- It is a Ruby-based Framework
- It Implements the MVC pattern

Let us have a look at these points in detail.

## RoR is a Ruby-Based Framework

The dynamic and open-ended nature of Ruby makes it an attractive option to build frameworks. Given the ease of meta-programming and reflection, blocks and iterators along with the exception handling, you have a language that could service any tier of a web application. That's what Mr. Hansson did. He took the different services provided by Ruby and created RoR out of it.

How Ruby eases the meta-programming is evident from Active Record, the ORM framework within RoR. Based on the name of the class, RoR (basically Ruby constructs) reads the schema and creates the objects of the class based on the data retrieved from the table on-the-fly. An `action` method communicates with the corresponding view using an instance variable and not through the explicit usage parameters sent through request objects or session objects. Apart from these, RoR makes heavy use of hash like structures and anonymous code blocks to reduce configuration.

Now that you have had a taste of how RoR makes use of Ruby, let us go the next aspect of RoR.

## RoR Implements MVC Pattern

You will have definitely heard the term MVC being used with different frameworks. But what is MVC and how is RoR concerned with it? MVC is a design pattern that provides a clear-cut demarcation between three aspects of an application—data access logic, the control flow logic, and the presentation logic. These three aspects have been deemed as M, V, and C. They stand for:

- Model—It represents the data processed by the application. It provides a link to the persistent storage (data store).
- View—the logic corresponding to the display of the data held by the Model is provided by the View. It is the only aspect of MVC that directly interacts with the user.
- Controller— It represents the control flow logic. The decisions about which View has to be called to display the current data, which part of the Model has to be updated are taken care of by the Controller. It sits at the boundary of your application and intercepts each request. It then calls the corresponding Model to update or retrieve data, and then chooses the appropriate View to display the data.

Coming to the question of how RoR is concerned with it, RoR implements MVC by providing three layers or components as a part of the framework. They are:

- Active Record
- Action View
- Action Controller

Action Controller and Action View together are known as Action Pack. Understanding more about these components will help you in not only finding an answer to the question of RoR's implementation of MVC, but will also tell you a great deal about how Ruby forms the base of RoR. So here it goes.

### Active Record

Active Record is the 'Model' in RoR. The Model component stores data and provides functionality to work with the data. Apart from being the Model component, Active Record is also an ORM framework. ORM stands for Object Relational Mapping. Hence, Active Record does the following, which constitutes functionalities of both a Model as well as an ORM framework:

- **Table to Class Mapping**: Each table is mapped to one or more classes. This is the default mapping, and the default mapping is based on convention rather than configuration. Having the name of the table plural and the name of the class singular is one of such conventions. As Active Record follows the Active Record data mapping pattern, the table attributes are mapped to the instance attributes at runtime. Hence, the classes don't need to provide the *getters* and *setters* for the table attributes, as you would have done in other ORM frameworks such as Hibernate. Once mapping is done, each object of the ORM class represents a specific row of the table with which the class has been mapped.
- **Database Connectivity**: You can connect to the database by making calls to the generic API that Active Record provides. The API then delegates the task to the database specific adaptor. In short, Active Record provides an abstraction over the process of connecting to a database. Active Record has adaptors for MySQL, Postgres, MS SQLServer, DB2, and SQLite databases. The connection aspect of Active Record comes into the picture only when you are not using Active Record with RoR. Yes, you can use Active Record even for Ruby projects that don't need to be web-enabled. In the case of RoR, you have to provide the parameters for connecting to the database in the `database.yml` file that resides in the `config` director of your RoR application. YML provides a way to describe data in a structured manner.
- **CRUD Operations**: CRUD stands for create, retrieve, update, and delete operations on a table. In terms of SQL, it means insertion, selection, updation, and deletion operations that are the basis of any application capable of saving user's choices. Because Active Record is an ORM framework, you always work with objects. To insert a new row, you will create an object of the class and populate its instance attributes with values. When a `select` statement is executed at the database side, Active Record creates objects corresponding to each row and provides them to you. Similarly, when an object is deleted, the corresponding row is also deleted. For retrieving existing data, Active Record provides a `find()` method. Thus, you can perform CRUD operations without worrying about the vendor of the database or the variation of SQL to be used.
- **Data Validation**: Validating the data before persisting it to the database is the first step in ensuring security of your website. To make it easier, Active Record provides validation of the Model component, also known as the Data Model. Data can be validated automatically when saved. You can also ensure that data is validated after an object is created or values are updated by using `validate_on_create()` and `validate_on_update()` methods. All the validation methods need to be overridden.

To create a model object from a table, you have to give the following command at the prompt of your application's directory:

```
ruby script/generate model <tablename_in_singular>
```

For example, to create model from a table named `Tale`, you have to issue the following command at the prompt:

```
c:\InstantRails\rail_apps\test_app\> ruby script/generate model Tale
```

## Action View

View component encompasses the logic for the presentation of the data present in the Model component. Action View is the View component of RoR. The functionalities provided by Action View range from template creation to Ajaxifying the web page. The most often used functionalities of Action View are:

- **Templates**: Templates are the files containing placeholders that will be replaced with content or expanded at runtime. The basic template functionality is provided by RHTML templates. They are HTML files in which you can embed Ruby code. When the template is called either through the browser or by the Controller component, the RUBY code is executed and its result is placed in place of the code itself. The final output is then sent to the browser. In RHTML templates, you can embed any kind of Ruby code.
- **Form Helper**: Even though you can embed Ruby code in RHTML, large chunks of code make the page unreadable and less maintainable. So you can place the code in Ruby files and call the relevant functions from within your RHTML page. This process is made simpler through Helpers. One of the commonly used Helpers is the Form Helper. It provides methods to create form elements such as checkboxes and textboxes. While using Form Helpers, you should start placing the element methods in the form of `form_tag()`.
- **Formatting Helper**: Formatting of the data to be presented is one of the major concerns in the presentation logic. Using Formatting Helpers you can easily format the data in the way you require it. It contains Helpers for the formatting of date, currency, and string.
- **Layout**: Layout defines how the various contents of a page are arranged. A dynamically created page may contain nesting of different pages. This can be the case even without using tables or frames. Action View helps in this case by providing a Layout service. Using the Layout API, you can create a page by nesting different pages. For example, if the pages of your site contain a header, body, and footer, then, only the body part will change with each page. By using Layout API, you can pass the content of the body without providing the header or body multiple times. Using this technique, you can even provide user-specific body content for a page.

A basic RHTML template can be created either by hand or through a command. The command is closely linked with the Controller component. We will look at it in the next section. Let's have a view of an RHTML file. As you already know, RHTML is an HTML file having embedded Ruby code. Let's say that you want to display a set of combo boxes, one each for day, month, and date, then the RHTML file will contain:

```
<html>
   <head>
       <title>Select Date</title>
   </head>
   <body>
       <h3> Please select the date of  the publication of the
       Tale</h1>
       date_select("post", "written_on", :start_year => 1855)
   </body>
</html>
```

You will see the details of this helper and many other such helpers in the coming chapters.

## Action Controller

The controller orchestrates the flow of logic. In a web-application, it is the Controller that regulates and orchestrates the flow of application logic. The controller sits at the boundary of an application and intercepts all the requests. Based on the request, it updates the corresponding Model object and calls the View logic to display the updated data. In RoR, the Action Controller provides the functionalities of the Controller. The main functionalities provided by the Action Controller, apart from the flow control logic, are:

- **Session Handling**: A session is the time period spent by a single user at a website. A session can be tracked in two ways—by cookies or by using a session object. Cookies are small files saved at the client side either for a definite time period such as a day or a week, or for the period of the user's stay at the website. The file contains the required user information. You can use a cookies object which is a hash like object to track the user session. However, cookies cannot hold objects. For that you need to use the session object. The session object is also a hash like structure that can store objects. By using cookies and session objects you can track the user session.

- **Filtering**: There are situations where you would like to call a particular set of statements before executing the logic in the Controller. Logging, user authentication and providing personalized response based on user are examples of such situations. To handle these situations, the Action Controller provides filters. There are three main filters—*before*, *after*, and *around*. These filters work as their name suggests. The *before* filter is executed before the logic within the Controller is executed. The *after* filter is called once the Controller logic is executed. And the *around* filter is a combination of the *before* and *after* filters. They wrap around the complete logic. So they are called before the execution as well as after the execution of the Controller logic.
- **Caching**: Caching is the process in which the most requested content is saved in a cache so that it need not be generated again and again. By using the Action Controller you can implement either page caching or action caching. Actions are the methods within the Controllers that correspond to a particular URL. In page caching, once generated, the content is not regenerated for the next request having the same URL. Instead, the already generated content is sent to the user. This is simple caching. The catch here is that the filter, if any defined, is executed only once. If you want to ensure that the filter is executed every time along with having the caching functionality, then you will have to use the action caching. It ensures that even though the content is not regenerated, the filter is executed for each request.

To create a controller you have to give the following command at the prompt:

```
ruby script/generate controller <controller_name> [<view_name>]
```

For example, to create a controller named `AddTale`, you have to give the following at the prompt:

```
c:\InstantRails\rail_apps\test_app\> ruby script/generate controller
AddTale
```

The created Controller file is a Ruby file just like the files created for the Model. However, the methods contained within this file are action methods that are mapped at the Controller level to the View templates, using the hash like structures. So you don't need to provide the XML configuration file for mapping them.

That completes the concepts and components of RoR. Now let's have a look at an application that uses the basic concepts discussed here.

## Hello World—the RoR Way

'Hello World' is one of the programs that any person tries out when starting with a new language or framework. It gives the basic steps in writing and executing the program in a successful way. The 'Hello World' of RoR is going to do a simple task—the Controller will pass a string to the View and the View will format it and present it to the user. There are four steps:

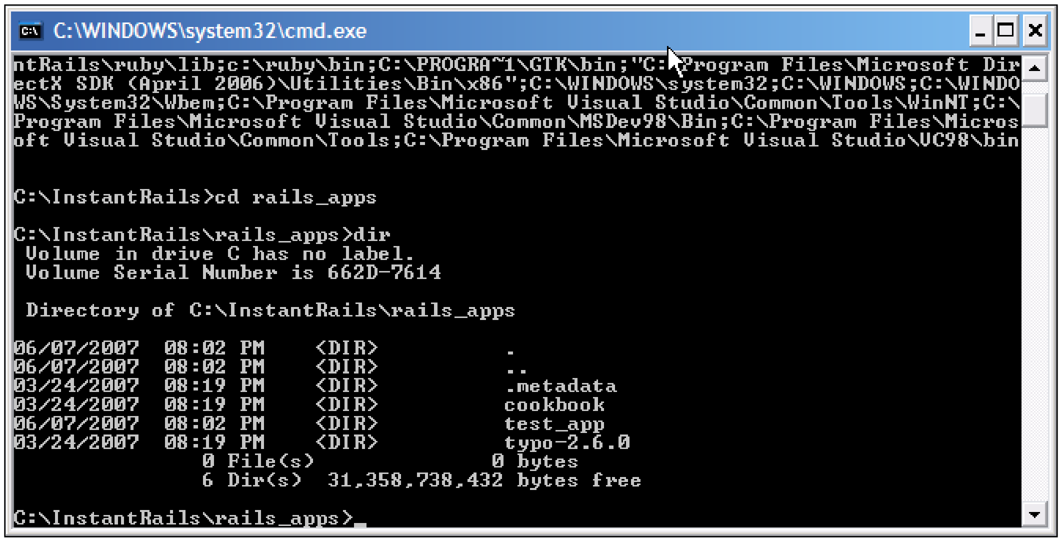
1. Setting up the Application Structure
2. Adding the First Controller class
3. Implementing the Action Method
4. Adding the View Template

## Setting up the Application Structure

Fire up the command prompt and give the following command:

```
C:\>use_ruby
```

It will drop you in the `rails_app` directory as shown below:

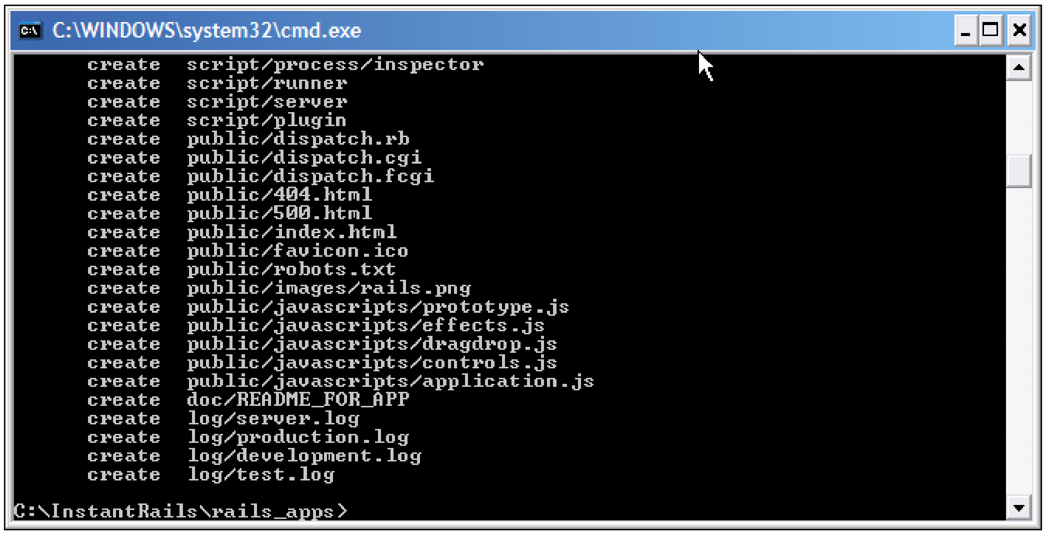


```
ntRails\ruby\lib;c:\ruby\bin;C:\PROGRA~1\GTK\bin;"C:\Program Files\Microsoft Dir
ectX SDK (April 2006)\Utilities\Bin\x86";C:\WINDOWS\system32;C:\WINDOWS;C:\WINDO
WS\System32\Wbem;C:\Program Files\Microsoft Visual Studio\Common\Tools\WinNT;C:\
Program Files\Microsoft Visual Studio\Common\MSDev98\Bin;C:\Program Files\Micros
oft Visual Studio\Common\Tools;C:\Program Files\Microsoft Visual Studio\VC98\bin


C:\InstantRails>cd rails_apps

C:\InstantRails\rails_apps>dir
 Volume in drive C has no label.
 Volume Serial Number is 662D-7614

 Directory of C:\InstantRails\rails_apps

06/07/2007  08:02 PM    <DIR>          .
06/07/2007  08:02 PM    <DIR>          ..
03/24/2007  08:19 PM    <DIR>          .metadata
03/24/2007  08:19 PM    <DIR>          cookbook
06/07/2007  08:02 PM    <DIR>          test_app
03/24/2007  08:19 PM    <DIR>          typo-2.6.0
               0 File(s)              0 bytes
               6 Dir(s)  31,358,738,432 bytes free

C:\InstantRails\rails_apps>_
```

Next, give the following command:

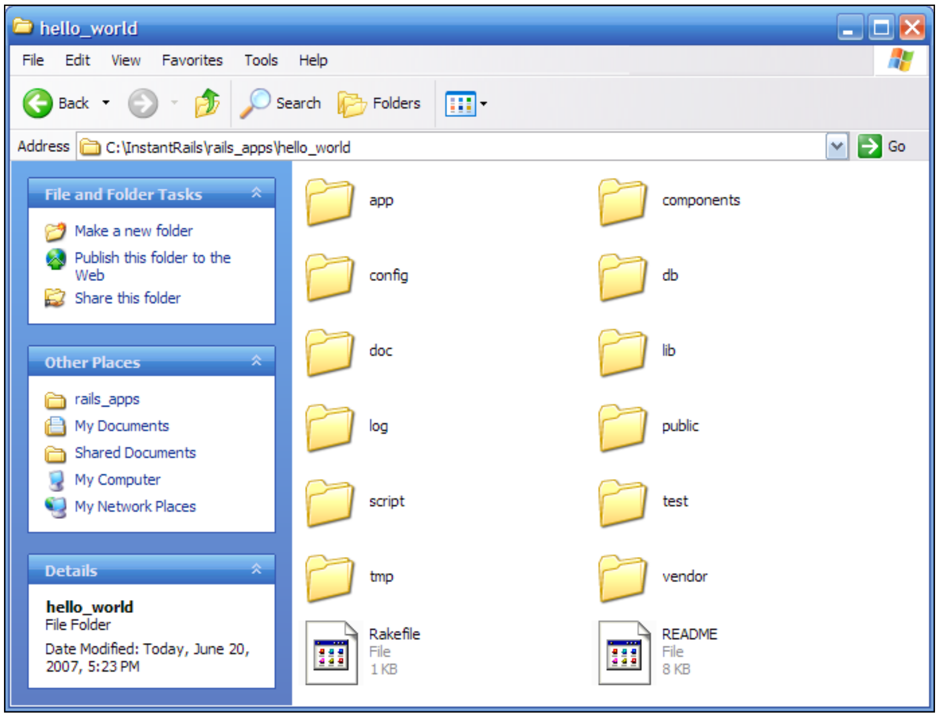
```
C:\InstantRails\rails_apps>rails hello_world
```

You will see the following screen:



```
        create  script/process/inspector
        create  script/runner
        create  script/server
        create  script/plugin
        create  public/dispatch.rb
        create  public/dispatch.cgi
        create  public/dispatch.fcgi
        create  public/404.html
        create  public/500.html
        create  public/index.html
        create  public/favicon.ico
        create  public/robots.txt
        create  public/images/rails.png
        create  public/javascripts/prototype.js
        create  public/javascripts/effects.js
        create  public/javascripts/dragdrop.js
        create  public/javascripts/controls.js
        create  public/javascripts/application.js
        create  doc/README_FOR_APP
        create  log/server.log
        create  log/production.log
        create  log/development.log
        create  log/test.log

C:\InstantRails\rails_apps>
```

That completes setting up the structure application. If you change to the directory of `hello_world` you will see the following structure:



The directory of our concern for the "Hello World" application is the **app** directory. It contains the following directories:

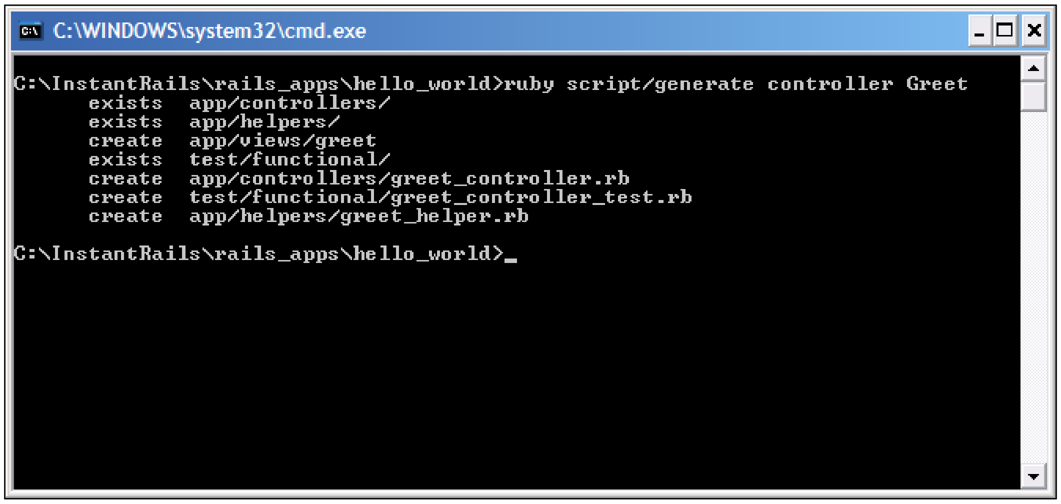
- **controllers**: It will hold all the generated controller files.
- **helpers**: It will contain the custom helpers.
- **models**: It will hold the generated model related files.
- **views**: It will have the template and presentation logic related files.

## Adding the First Controller Class

At the command prompt, give the following command to generate the Controller named `Greet`.

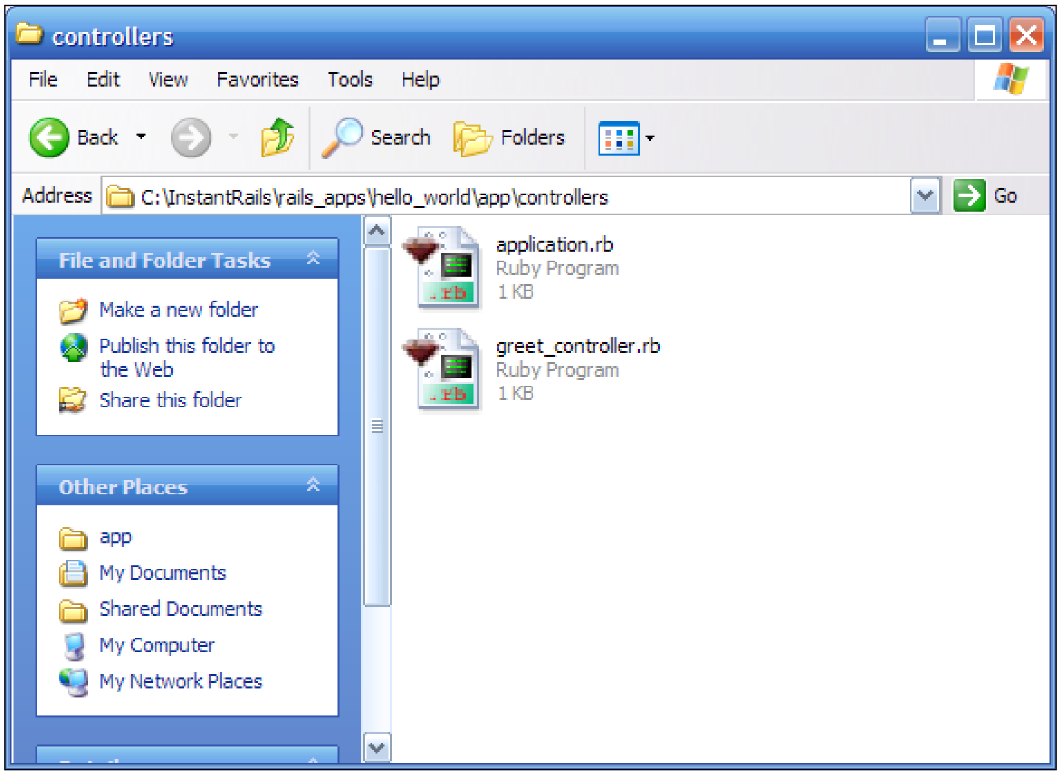
```
C:\InstantRails\rails_apps\hello_world>ruby script/generate controller
Greet
```

You will get a bunch of messages as shown in the following screen:



```
C:\InstantRails\rails_apps\hello_world>ruby script/generate controller Greet
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/greet
      exists  test/functional/
      create  app/controllers/greet_controller.rb
      create  test/functional/greet_controller_test.rb
      create  app/helpers/greet_helper.rb

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

If you look in the `app/controllers` directory you will see the files as shown in the following image



Of the two files you will be working with the `greet_controller.rb` file. We will be defining the `action` method in that file.

## Defining the Action Method

Every Controller contains actions which are methods that are mapped to a particular URL. So, whenever there is a request for the URL, the corresponding action is called and it is executed. We will be adding the action method `index`. To do so, open the `greet_controller.rb` file. It will already contain the following method:

```
class GreetController < ApplicationController
end
```

The class `GreetController` has been inherited from *the* `ApplicationController`. The action method `index` will set an instance attribute named `opening_lines`. To the class add the following code:

```
class GreetController < ApplicationController
   def index
   @opening_lines="Walking down the memory lane, Standing at the
   arch of time"
   end

end
```

That completes this step. Next, let us add the View template.

## Adding the View Template

Views are mapped with the action methods. Since our action method is named `index`, the name of the RHTML template will be `index.rhtml`. Now you can see the philosophy of Convention over Configuration at work. You don't need to add an entry into any configuration file as you would have done in frameworks such as Struts or JSF. Here, when RoR sees a URL ending with the action index, it will call up the action method named `index`, execute it, and the give the result to the template having the same name (`index.rhtml`). It searches for the template in the `app/views/greet` folder. So you have to create the `index.rhtml` in the `apps/views/greet` directory.

Open your favourite editor and write the following code:

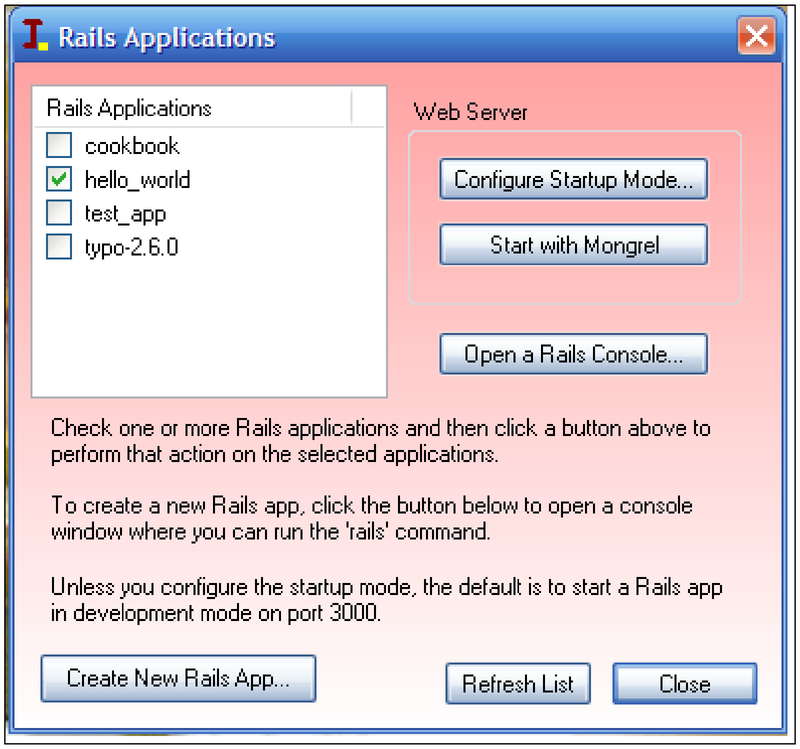
```
<html>
<head>
<title>Hello World from RoR</title>
</head>
<body>
<h2>Greeting From RoR</h2>
<br/>
Todays's Opening Lines are:<br/>
<%=simple_format(@opening_lines)%>
</body>
</html>
```

The `simple_format` helper formats the given text preserving breaks and new lines. The `<%=/%>` tags output the result of the expression given between them.

Save it as `index.rhtml` in the `app/views/greet` folder. That completes the 'Hello World' application.

## Testing the Application

Start the Instant Rails Manger application and select **I | Rails Applications | Manage Rails Applications...**. In the window opened, select the **hello_world** checkbox and click on **Start with Mongrel**.



Check one or more Rails applications and then click a button above to perform that action on the selected applications.

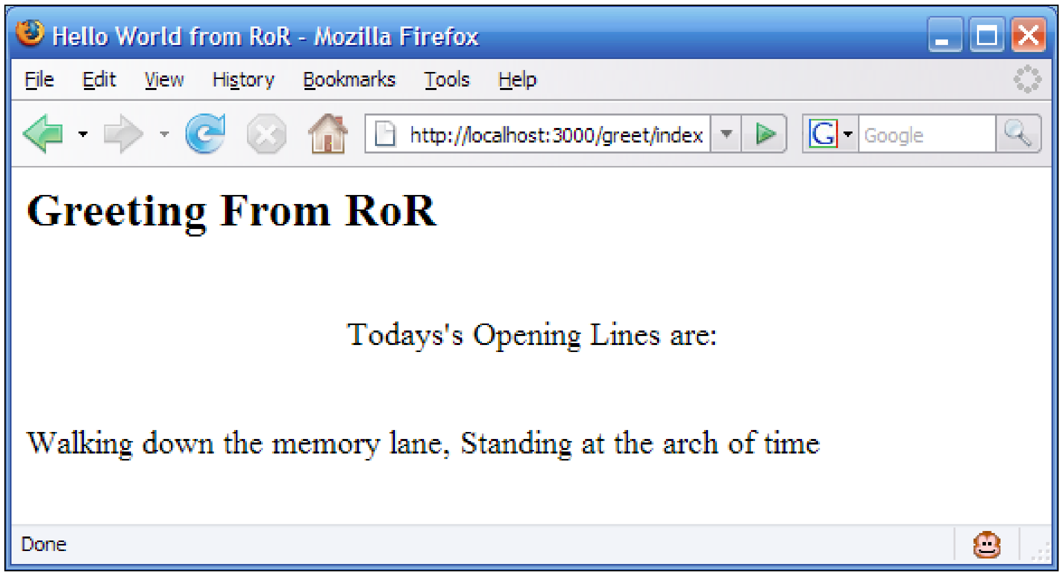
To create a new Rails app, click the button below to open a console window where you can run the 'rails' command.

Unless you configure the startup mode, the default is to start a Rails app in development mode on port 3000.

Once the Mongrel server starts, open the web browser and give the following URL:

```
http://localhost:3000/greet/index
```

If you get the following screen, then everything is well and fine.


Hello World from RoR - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/greet/index
Google
Done

Greeting From RoR

Todays's Opening Lines are:

Walking down the memory lane, Standing at the arch of time

One thing you have to keep in mind is that when Mongrel is started in the development environment, the name of application need not be given before the controller's name in the URL.

## Summary

This chapter introduced you to the basics of Ruby including classes, attributes, methods, and blocks, as well as the main concepts and components of RoR. We have also seen how to set up the application structure. In the next chapter, the development of the TaleWiki application starts, where these concepts will help you in building the application. So get ready for RoR in real world.

# 3
# TaleWiki—The Basic Setup

In the last two chapters, you saw what RoR provides and how to access the services of RoR to ease the path of dynamic website development. However, the chapters dealt primarily with the theoretical aspect of RoR. Now is the time to apply the theory to develop the real world application. That's what we are going to do from this chapter onwards. Beginning with this chapter, we will be developing a website called TaleWiki. Each chapter will add a new functionality or enhance the existing functionality.

In this chapter, you will understand the basic requirements of TaleWiki. We will then move on to design and set up the database for these requirements. The next step will tell you about developing the website and we will wrap up the chapter by testing TaleWiki with its bare minimum functionality. So let's get started.

## Understanding the Requirements

The requirements of a software system define and set the boundaries of functionalities provided by that system and expected output in terms of reports. In our case, system requirements will tell us the expected services that TaleWiki will provide. The requirements can be broadly classified into two sets:

- Overall System Requirement
- Module-Specific Requirements

The former will tell you about the features and functionalities to be provided by TaleWiki. And the latter will detail the individual functionalities planned in the overall system requirement. The overall requirement will also help you to keep track of the functionalities as the system evolves. In short, the overall requirement lays out the modules of the system, and the module-specific requirements go in depth into the functionalities of the modules. Each chapter will go into the details of the modules and their functionality.

## System Requirements

Before building any kind of system, it is a good practice to understand what its boundaries are. TaleWiki is no exception. The first thing to decide is what lies within the boundary and what stays out. TaleWiki, as the name suggests, is about tales or stories. But that's just the first part. 'Wiki' suggests a collaborative environment. Collaboration comes in many forms, such as comments, tagging, and user groups—all of these are part of the collaboration. It also means interaction between the users. Based on these 'starting inputs', we can definitely say that TaleWiki will provide following functionalities:

- **Managing the Stories**: Stories or tales form the core of the system. So all the operations including creation, updation, and deletion of the stories need to be supported.
- **Managing the Users**: It is the users who provide the stories. So users need to be managed. The management of users not only includes the tasks of registering and providing functionality to update their details, but also tasks such as assigning the roles, authenticating, and authorizing them come into picture. In short, the user management also includes role management and user authentication/authorization functionalities.
- **Gathering Comments**: Feedback is what keeps the stories coming. Once a story is published, the comments need to be collected from other users so that the author can have an understanding of the opinions of his/her target audience. Comments can also take the form of an impromptu discussion about a particular issue raised by the particular story or tale.
- **Tagging the Stories**: Tags increase the usability of a system by providing a better way to navigate and find what one needs. TaleWiki will provide tagging of the individual stories so that users can easily find what they need.
- **Providing the Administrative Interface**: A site always needs to be administered. The administrative tasks can include knowing the number of active users or banning a particular user. TaleWiki will provide an easy-to-use interface to do the administrative stuff.

The functionalities listed above just provide the overview. Each item of the list can be treated as a module, thus needing to be handled separately. The 'separate handling' starts from here on.

## Module-Specific Requirements

Once the modules based on functionalities have been decided, the next step is to go into the depth of each of the proposed modules and set the boundaries for implementation. This will help in understanding the sub-modules or tasks that form the module as well as know which tasks require communication with other modules. So let's get started.

## Managing the Stories

This is the basic functionality of TaleWiki. A story can be anything from fiction and non-fiction to current affairs. There are four main tasks with any kind of story. Firstly, they need to be written and saved with the system, then the story may need updating, and finally be posted for reading. A situation may arise where the user or the administrator may want to delete a story. Based on these tasks we can definitely say that the Story Management module can be divided into the following sub-modules or tasks:

- **Submitting the Story**: The first step is to submit a story and save it with the system. The interface for this sub-module will have a provision to enter the details of the story. The details will include genre, author or source, and publication date. Of these, publication date will typically be the date of submission of the story.
- **Updating a Submission**: This is a usual scenario—you have submitted a post and then you remember that you have missed some important point. That's where this task or sub-module comes into picture. It will provide an easy way to update a submitted story. All the details of the submission will be updatable by the user except the publication date and genre. Whenever a user updates his/her story, the status field will be updated to reflect that revisions have been done to the story. Once the status has been changed, the UI will also reflect it by making the status visible to the reader (other users).
- **Deleting Submissions**: When a submission is deleted then it will be deleted from the database itself.
- **Publishing a Submission**: Once a story has been submitted, it needs to be published so that other users can read it. The publishing sub-module will provide two views—*list* and *detailed*. The *list* view will provide all the submissions as a list, and the *details* view will show the submission in its entirety. The *list* view will not contain the story itself; it will contain only the title, submission date, genre, and the author/source information. The *detailed* view will show one story at a time with all the information provided in the *list* view along with the story itself and comments added by the user. Because the *detailed* view will contain the comments, its implementation will be dealt in Chapter 5 where we will be *Developing the Comments Management Module*.

Genre is one of the related information of a story. However, the 'acceptable values' must be provided by the administrator. The user can only choose from the provided list of genres. To achieve this end, TaleWiki needs to manage the genre as well. So genre management—part of the story management—supports the tasks related to managing genres. Genre management will enable the administrator to perform the following tasks:

- **Adding new Genre**: When a new genre has to be added, the admin can use the interface provided by this sub-module. The admin can enter details including the name of the genre and a small description about the genre.
- **Modifying an Existing Genre**: The task of modifying the details of an existing genre can be carried out by using this sub-module. Any modification done using this module will be reflected in the stories.
- **Deleting a Genre**: The Admin may want to remove a genre. To do this, this sub-module will provide an interface to the admin.
- **Viewing the List of Existing Genres**: The view will be in the form of a list only. It will show the genre and its related description.

That completes the details of functionalities that are going to be provided by the Story Management module. Let us move on to the next step—designing the database where the stories and their related information will be stored.

## Designing the Database

The next step is to design the underlying database and the tables that will act as a data store for the Story Management module. As this is the first time that we are going to discuss the database design in the context of RoR, we will not only be discussing the table design but also the conventions that need to be followed. The steps in designing the database and tables are:

1. Understanding the Conventions
2. Designing the E-R model
3. Defining the Schema
4. Creating the Tables

The first step is one-time as it is about the conventions that you need to follow while designing the database and tables for use with Active Record. You will be repeating the other steps for each module.

## Understanding the Conventions

The philosophy of Convention-over-Configuration is most evident in the rules for the database design. These conventions/rules bring down the configuration factor to near-zero. Why only near-zero? The reason is that the database connection parameters—such as the type of database, hostname, or user name need to be specified in `database.yml`. Before going into the details of the configuration aspect let us have a look at the conventions for the database and tables.

- **The name of database should be suffixed wit**h **_development**, **_test, or _production**. By using these suffixes, RoR knows whether the application is in development mode, testing mode, or production mode. The mode helps the application to access the corresponding database. One thing you have to keep in mind is that unlike development or production mode databases, the test mode database is re-created every time the application starts or restarts. Based on this convention, the database name of our application will be **talewiki_development** during the development cycle.
- **The name of the table must be in its plural form**: The table name must be pluralized. That means if you want to name the table for story *tale*, it must be named *tales*. If you want to use a name with multiple words, you have to delimit them with *underscores* (_). The beauty of Active Record is that if you give a name such as *axes* (plural of *axis*), Active Record will derive the singular name for the corresponding class automatically.
- **The Primary Key should always be named 'id'**: Having the primary key named id removes two problems. The first is that you don't need to think about a new name for the primary column and tell Active Record that this particular column is the Primary Key. To understand the second advantage, assume that for the user table you have used the SSN as the Primary Key. And if in the future the length of the SSN changes, not only you will have to change it in the user table but also in all those tables where the SSN is used as foreign key reference. This will mean a lot of work and if this happens for a deployed and widely-used application then it will lead to the sleepless nights. By keeping the Primary key as id, this problem is completely avoided. The only thing you have to keep in mind is that the id is an attribute in addition to other attributes. To use an example, you will still have to use SSN in the user table, but it will be constrained to the user table only.
- **The Data type of the Primary key should be Integer**: Active Record assumes that the data type of the Primary Key is of type integer or long. This helps in easier generation of id values automatically.

- **The Foreign keys should be named 'name of the referenced table in singular _id'**: Whenever you want to provide a foreign key, the attribute should be named with the _id suffix of the name of the table in singular. For example, if *tales* table needs to refer the *genres* table, then the *tales* table should contain an attribute named *genre_id*.

These conventions can be overridden but not recommended. Next, let us look at the E-R model.

## Designing the E-R Model

The first step in designing a database is coming out with the Entity-Relationship model for the given scenario. This will help you in arriving at tables and give you a clear-cut picture of the relationship between the tables. Let us start with the scenario.

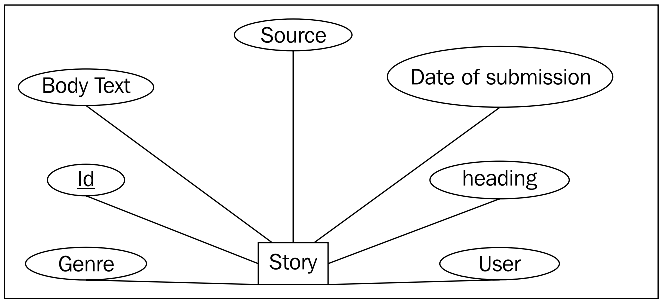
The Story Management module manages stories posted by the user. Each posted story contains the following information:

- Id that uniquely identifies a story
- The heading of the story
- The body or the complete story
- The day the story was submitted
- The author or the source of the story
- The genre or category of the story
- The status of the story—new or updated.
- The name of the user who posted the story

Here, the user and the author are different, as the author may be a different person and the user may be a person who submits it. In most cases, the source will be more appropriate. Based on this information we can arrive at the first entity and its attributes. The first entity is **Story**. Its attributes are:

- **Id**—this is the Primary key attribute as it can uniquely identify a story
- **heading**—the title of the story
- **Body text**—the body of the story
- **Date of submission**—the day the user submitted the story
- **Source**—the source from where the story was found. If it is written by the user himself/herself, the source will be the user's id
- **Genre**—the category of the story
- **User**—the user who submitted the story

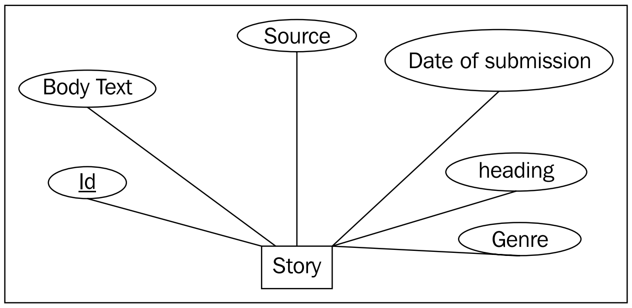
Diagrammatically, the entity will be as follows:



Of these, there are two attributes that need a special attention—User and Genre. The reason is that they provide extra information about the Story, but they are not completely dependent on the Id of the Story. And even if the Story is deleted, the information related to these attributes may be required in the future. In short, User and Genre can be separate entities themselves. Keeping this point in mind, if we look at the entities, we have:

- Story
- Genre
- User

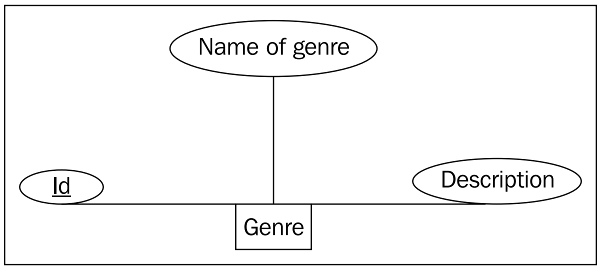
We will be getting back to the User entity in the next chapter. So for this chapter, we have two main entities—Story and Genre. To understand why Genre has to be a different entity and not just an attribute of the Story entity, consider this scenario: A user with Id 'John Doe' submitted a story for the genre named 'Obscure news'. No other user has submitted any story for this genre. So the genre becomes unique. If the user, in the future, deletes that story, the genre named 'Obscure news' will also be deleted. So even if there had been a genre for submitting news with little value or limited interest, the entry no longer exits. Such a scenario can be repeated with different types of stories. In order to avoid such situations, we are making **Genre** a separate entity. The revised **Story** entity is as follows:



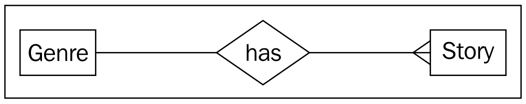
Now that the 'whys' of making Genre a separate entity have been dealt with, let us see what will be the attributes of this entity. Each **genre** will require the following:

- **Id**—it will separate out the different genres
- **Name of genre**—the name of the particular genre
- **Description**—a short description about the genre.

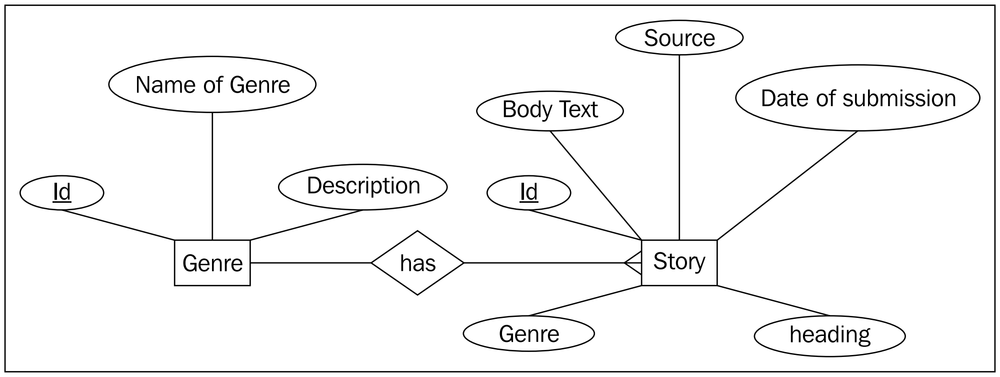
The **Genre** entity with all its attributes will be as follows:



We have identified the two entities that form the basis of the Story Management module. The question that now arises is, 'Is there any relation between these two entities? If yes, what type of relation is it?' The answer to the first question is a definitive yes. If it was not so, the Story entity won't be having an attribute named Genre. The answer to the next question lies in another question—'Does a story have many genres or does the Genre have many stories?' We can definitely say that a story is not going to have multiple genres. So, one story having many genres is out of the picture. However, one genre can have multiple stories. So the relationship is one-to-many when viewed from **Genre** to **Story**. If you want to view it the other way, we can say that the relationship is many-to-one from **Story** to **Genre**. The complete entities with relationship are shown as follows:



The following is the complete E-R diagram:



That completes the E-R design step. The next step is to derive a schema from the E-R design. That's what is coming up next.

## Defining the Schema

The next step is to define the schema based on the E-R model. From the E-R model, we have three things:

- The two entities—Story and Genre
- Their attributes
- The relationship between them which is many-to-one

Because the relationship is many-to-one, there is no requirement of converting the relationship into a table. So we have two entities that can become tables—Story and Genre. In defining a schema based on the E-R model, what you have to do is to provide details for each of the attributes. These details include the data type of the attribute, length of the acceptable value of the attribute, and a description about the properties. Describing the properties will be overkill as it has been already done. We will be defining the schema for the Genre entity.

| **Name of the attribute** | **Data type of the attribute** | **Length of the acceptable value** |
|---|---|---|
| Id | Integer | 10 |
| Name of Genre | Varchar | 25 |
| Description | Varchar | 100 |

The **Id** can have value up to 10 numbers. The **Name of Genre** can accept values up to 25 characters, and **Description** can be of 100 characters. These are standard SQL-types and not specific to any database.

The schema for the Story entity is as follows:

| Name of the attribute | Data type of the attribute | Length of the acceptable value |
|---|---|---|
| Id | Integer | 10 |
| Title | Varchar | 100 |
| Body Text | Varchar | 1000 |
| Date of Submission | Date | |
| Source | Varchar | 50 |
| Status | Varchar | 15 |
| Id of Genre | Integer | 10 |

The main point to be noted here is the presence of the **Id of Genre** attribute. The reason for its presence is the many-to-one relationship between the Story and Genre. The relationship is converted to the foreign key reference. Each row of Story will have an Id of Genre which needs not to be unique. That completes the schema design. In the next step we will derive the tables for the MySQL database from the schema designed just now.

## Creating the Tables

The schema defines the tables in a Database-server-independent manner. That means the data-types we used were not specific to a particular database server such as MySQL or Oracle 9i. However, while creating tables the database-independent types will have to be substituted with the database-server-specific data type. Not only that, each database server gives its own flavor to the Data Definition Language (DDL) or table creation queries. MySQL is the database server that we are going to use.

In MySQL, you can specify relations if you are using the InnoDB engine. A simple table creation statement in MySQL looks like:

```
Create table <table_name>(
   <column_1> <data_type>(length),
   :
)engine=INNODB;
```

However, before creating the tables, you will have to create a database so that the tables can be created within it. The name of our database will be talewiki_development. To create this database, give the following command on the MySQL prompt.

```
mysql>create database talewiki_development;
```

MySQL will respond with:

```
Query OK, 1 row affected
>
```

Next, we have to tell MySQL that we will like to create tables in the taleswiki_development database. For that issue use the following command:

```
mysql>use talewiki_development;
```

MySQL will respond with:

```
Database changed
```

Now we are ready to create the tables. The first table is for Genre. Applying RoR's convention, the name of the table will be `genres`. The table creation statement is as follows:

```
CREATE TABLE `genres` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
`genre_name` VARCHAR( 25 ) NOT NULL
`description` VARCHAR( 100 ) NOT NULL
) ENGINE = innodb;
```

Keep in mind that the names of the fields are surrounded with back quotes and not single quotes. If you look at the `id` field, you will notice that we have not provided the length of the field. So, it is also a convention not to provide the field length. Once executed, MySQL responds with:

```
Query OK, 1 row affected
```

Next is the Story table. Here we will give a name different from that of the schema. Let us name it `tales`. The reason is that this table is the core table of TaleWiki. So it is appropriate that the name of the core table matches with that of the application. The table creation statement for `tales` table is:

```
CREATE TABLE `tales` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
`title` VARCHAR( 100 ) NOT NULL,
`body_text` TEXT NOT NULL,
`submission_date` DATE NOT NULL,
`source` VARCHAR( 50 ) NOT NULL,
`status` VARCHAR( 15 ) NOT NULL,
`genre_id` INT NOT NULL,
CONSTRAINT `fk_tales_genres` FOREIGN KEY (`genre_id`) REFERENCES
genres( `id`)
) ENGINE = innodb;
```

If the query has been executed successfully, MySQL will, as usual, respond with:

```
Query OK, 1 row affected
```

While creating the `tales` table, if the foreign key reference is having any error, MySQL will complain with error no. 102. Whenever you get an error no. 102, check the foreign key part of your `CREATE` statement. That completes the database design for this module. From the next section, we will be treading in the domain of RoR.

> The SQL data-types and their matching MySQL data-types are as follows:
> Integer becomes INT
> Varchar remains Varchar
> Varchar with length more than 500 can be mapped to Text

## TaleWiki—Developing the Tale Management Module

Though the heading of the section is *developing the tale management module*, we will be dealing with both *tale* (story) as well as *genre* management. We will be developing these modules step-by-step. The steps that we will be following are:

- Creating the Application Structure
- Generating the Scaffolds for genres and tales tables
- Customizing the Model
- Customizing the Controller
- Refining the View
- Testing the Application

Testing follows each of these steps. Of these, the first will be done only in this chapter as from the next chapter onwards we will be building upon the application. So, without any further ado let us get started.

### Creating the Application Structure

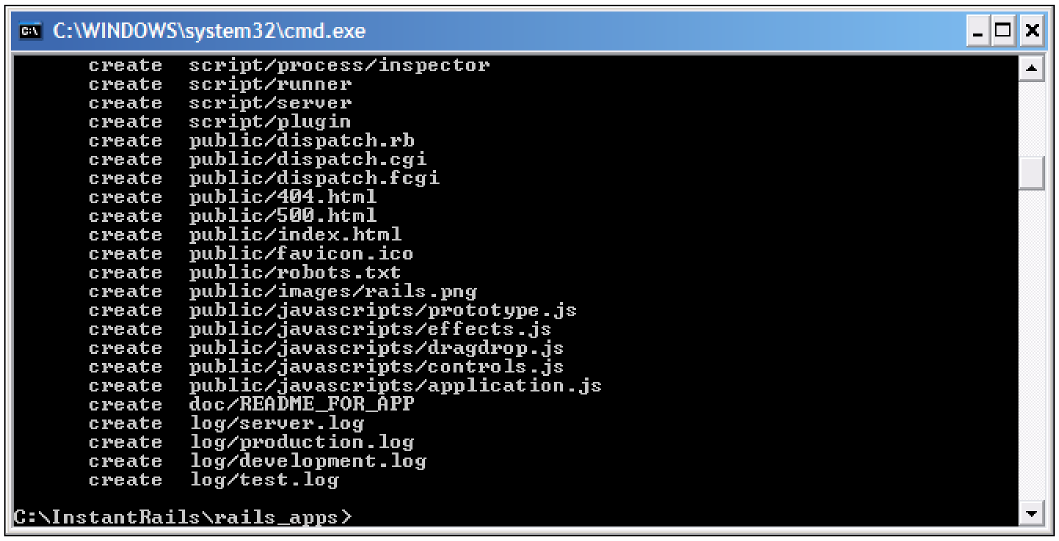
To create the application structure, fire up the command prompt and give the following command:

```
>use_ruby
```

Once the prompt drops you into the **rails_apps** directory, give the following command to create the application:

```
c:\InstantRails\rails_apps\ > rails talewiki
```

As a response you will get the following screen:



```
      create  script/process/inspector
      create  script/runner
      create  script/server
      create  script/plugin
      create  public/dispatch.rb
      create  public/dispatch.cgi
      create  public/dispatch.fcgi
      create  public/404.html
      create  public/500.html
      create  public/index.html
      create  public/favicon.ico
      create  public/robots.txt
      create  public/images/rails.png
      create  public/javascripts/prototype.js
      create  public/javascripts/effects.js
      create  public/javascripts/dragdrop.js
      create  public/javascripts/controls.js
      create  public/javascripts/application.js
      create  doc/README_FOR_APP
      create  log/server.log
      create  log/production.log
      create  log/development.log
      create  log/test.log

C:\InstantRails\rails_apps>
```

It's same as that we did in Chapter 2. However for the sake of continuity, it has been recounted. That completes our first step. In the second step, we will be generating scaffolds for the tables.

## Generating the Scaffolds

Before we generate scaffolds for our tables, it is better to understand what a scaffold is. In general terms, a scaffold refers to a temporary platform to provide support to the workers while working at heights above the ground. Here, the keywords are **temporary platform** and **provide support**. Therefore, scaffolds are temporary and they provide support to accomplish a particular task. If we take this concept and bring it into RoR, the meaning still holds true. We have a table, which we want to access by mapping it to a model and manipulating it through a view and corresponding controller. However, once done, we will be customizing it. For such a situation you have to use scaffolds. So, they are frameworks generated by RoR to provide access and manipulation interfaces that can be customized later. That's what scaffolds are. Let's generate scaffolds for our tables.

The first step in creating the scaffolds is to configure the connection parameters. To do it, open the `database.yml` file within the `config` directory. If it contains the following details, then you are ready.

```
development:
  adapter: mysql
  database: talewiki_development
  username: root
  password:.
  host: localhost
#Warning: The database defined as 'test' will be erased and
#re-generated from your development database when you run 'rake'.
#Do not set this db to the same as development or production.
test:
  adapter: mysql
  database: talewiki_test
  username: root
  password:
  host: localhost
production:
  adapter: mysql
  database: talewiki_production
  username: root
  password:
  host: localhost
```

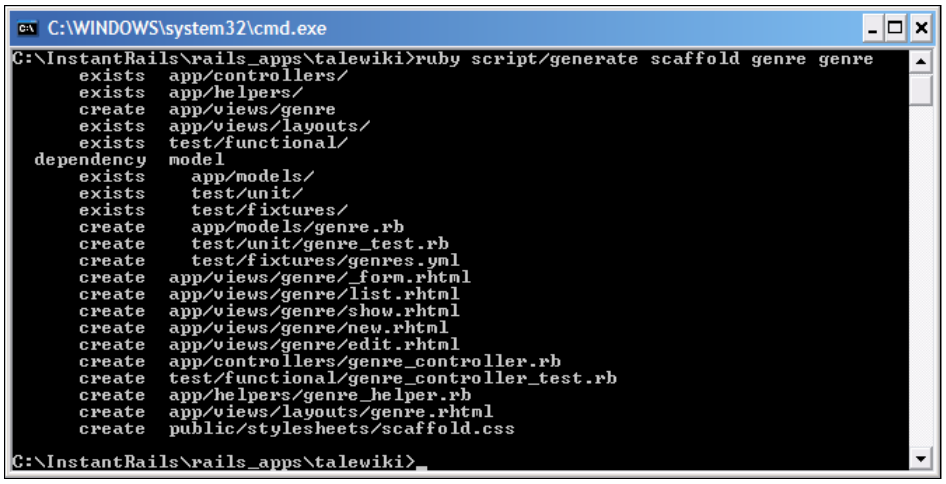
Next, we have to give the scaffold command to generate scaffold. Essentially, the scaffold command is the scaffold argument to the `script` named `generate`. The syntax of the command is:

```
ruby script/generate scaffold <table name in singular> <name of the
controller>
```

So to create the scaffold for the `genre` table change into talewiki directory and give the following command:

```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold genre
genre
```

RoR will respond with the following screen:



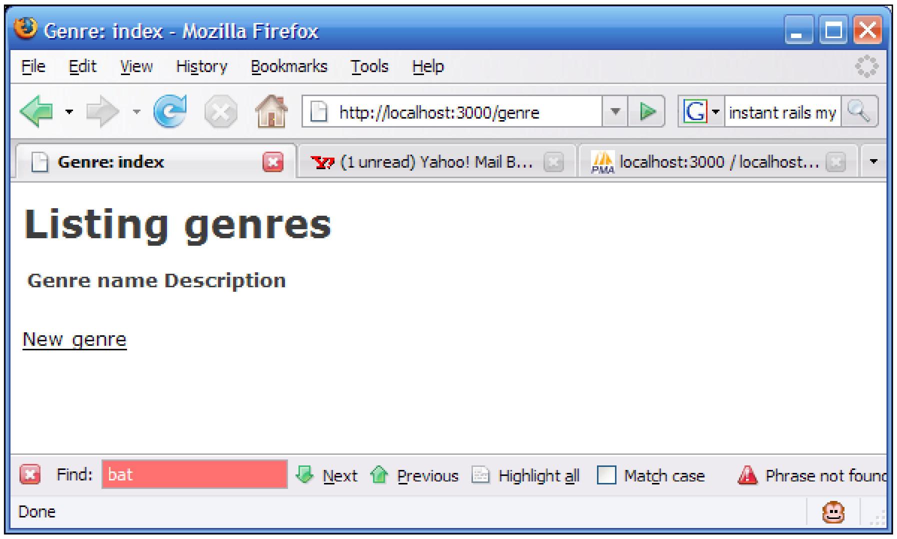
```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold genre genre
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/genre
      exists  app/views/layouts/
      exists  test/functional/
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/genre.rb
      create    test/unit/genre_test.rb
      create    test/fixtures/genres.yml
      create  app/views/genre/_form.rhtml
      create  app/views/genre/list.rhtml
      create  app/views/genre/show.rhtml
      create  app/views/genre/new.rhtml
      create  app/views/genre/edit.rhtml
      create  app/controllers/genre_controller.rb
      create  test/functional/genre_controller_test.rb
      create  app/helpers/genre_helper.rb
      create  app/views/layouts/genre.rhtml
      create  public/stylesheets/scaffold.css

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

Similarly, for the `tales` table give the following command:

```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold tale
tale
```

RoR will respond with a screen similar to that shown for `genre`. The next step is to test the generated scaffolds. To test the scaffold genre, start the mongrel server. Once it is started, start the browser and give the following URL:

`http://localhost:3000/genre`


Genre: index - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/genre
instant rails my
Genre: index
(1 unread) Yahoo! Mail B...
localhost:3000 / localhost...
Find: bat
Next Previous Highlight all Match case Phrase not found
Done

Listing genres

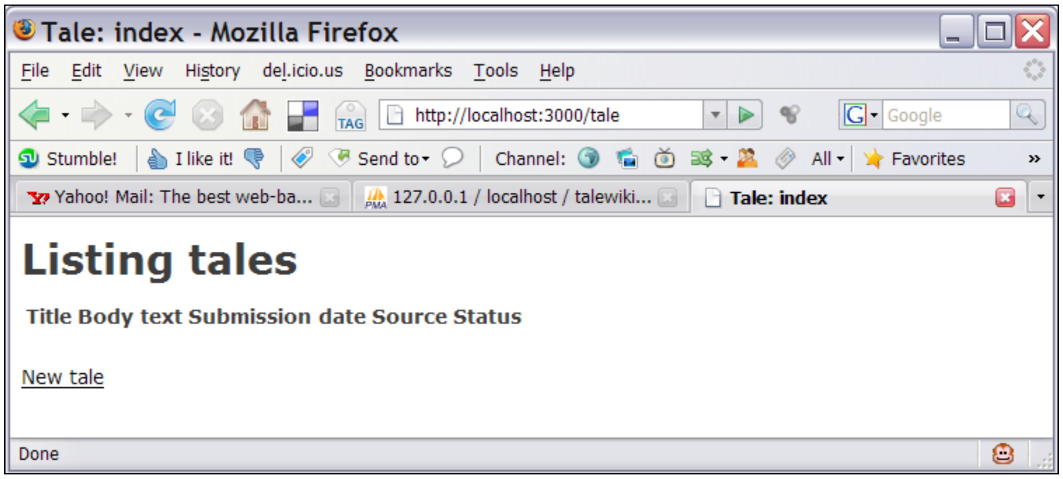
Genre name Description

New genre

If you get the screen just shown, then everything is fine. Next, to test the tale scaffold give the following URL in the browser:

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

If you get a screen similar to that shown next, then tale scaffold has also been generated correctly.


Tale: index - Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/tale
Stumble! I like it! Send to Channel: All Favorites
Yahoo! Mail: The best web-ba... 127.0.0.1 / localhost / talewiki... Tale: index
Done

Listing tales

Title Body text Submission date Source Status

New tale

That completes the scaffold generation. However, the logic generated by the scaffold doesn't meet all of our requirements. So we will be refining each component starting with Model. That is our next step.

## Customizing the Model

When the scaffold is generated, the Model component or the ORM classes mapped to the tables are placed in `app/models directory`. If you look at the directory, you will see two files—`genre.rb` and `tale.rb`, which correspond to genre and tale tables respectively. The models generated by the scaffold lack two important functionalities:

- Relationship Mapping
- Data Validation

We will be adding these two functionalities to the Model component.

## Relationship Mapping

In RoR, relationships can be mapped using the following declarations:

- **has_many**: It is used with the class which is at the 'one' end of the one-to-many relationship. In our case, we will be using it with the genre class.
- **belongs_to**: It is used to with the class which is at the 'many' end of the one-to-many relationship. We will be using it with the tale class.

To add relationship mapping to the `Genre` class, open the `genre.rb` file. You will see the following code:

```
class Genre < ActiveRecord::Base
end
```

Now we know that the genre will have many tales. To tell RoR the same, we have to add a `has_many` declaration to the `Genre` class. The argument to this declaration will be the tale class as it is at the 'many' end of the relationship. After adding the declaration, the code will be:

```
class Genre < ActiveRecord::Base
has_many:tales
end
```

Similarly, we have to tell that the `Tale` class is at the other end of the relationship. For that we will be using the `belongs_to` declaration with `genre` as its parameter. Open the `tale.rb` file. You will find the following code:

```
class Tale < ActiveRecord::Base
end
```

After adding the declaration, it will look like this:

```
class Tale < ActiveRecord::Base
belongs_to :genre
end
```

Here, in both the cases, we are not providing the class name as the parameter. Instead, we are giving the object names as the parameter. This is again a convention. An object can have the same name as the class with its starting letter in lower case. Thus, our model classes have become model aware. Next we will be adding the validation.

## Data Validation

One main point to keep in mind, while developing a database driven dynamic website, is the correctness of data. It is a common practice among users to leave fields empty. Validating whether the input is empty is one of the most common uses of data validation. It is also known as required field validation. If we look at TaleWiki, when the user submits the data for the new genre, the name of the genre should not be empty. Similarly, when the user submits a new story, the fields including the title of the story, its body, and the source should not be left empty. To achieve this end, RoR provides the `validates_presence_of ()` method. You just have to pass the fields to be validated. It will become clearer when we apply it to the genre and tale model.

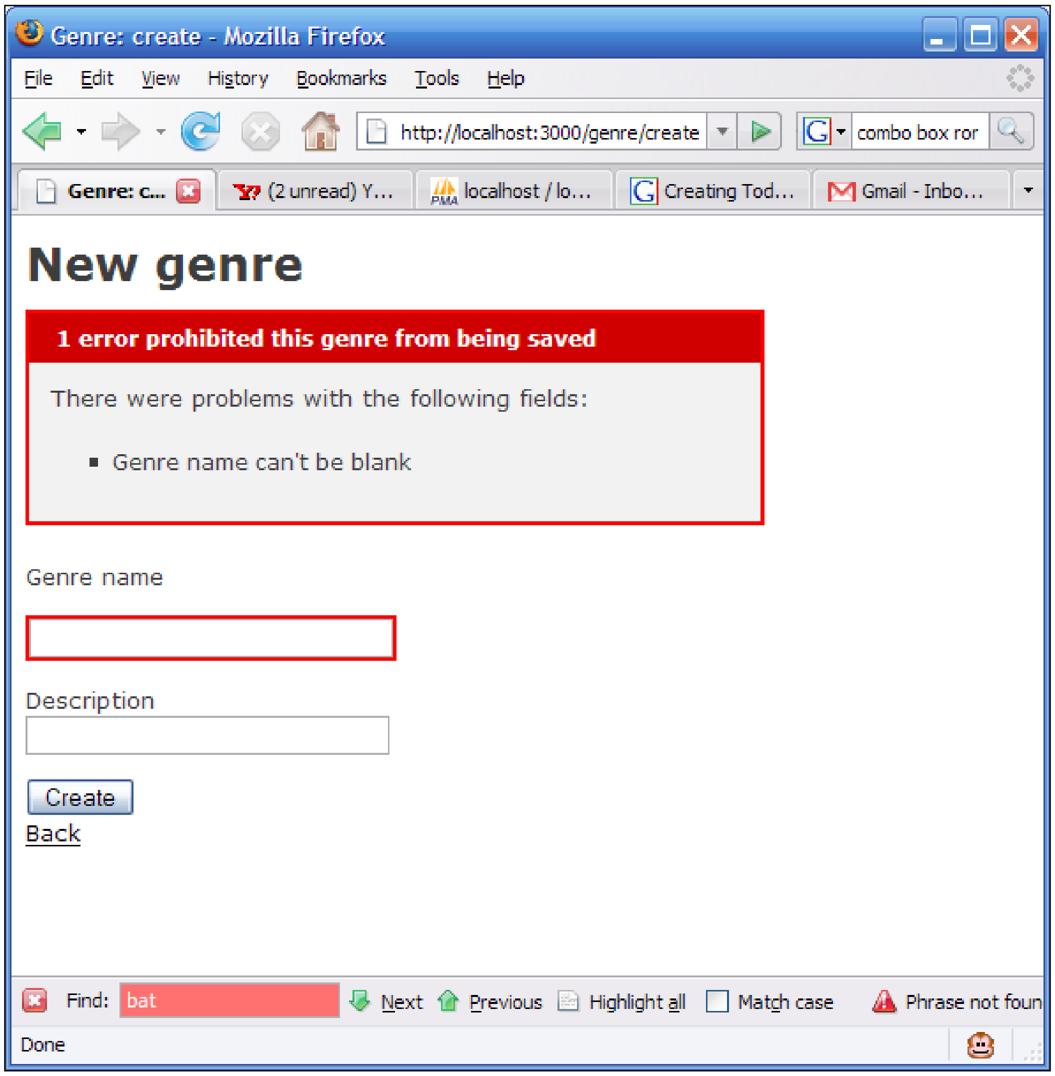
Let us take up `Genre` class first. We have to check the name field for null or empty values. To do it, open the `genre.rb` file and add to it the `validates_presence_of ()` method with the name `genre_name`. The Convention-over-Configuration principle is at play here. RoR assumes that the name of the field in the View is the same as that of the Model's attribute. After adding the method, the Genre class will look like as follows:

```
class Genre < ActiveRecord::Base
  validates_presence_of :genre_name
  has_many :tales
end
```

To test the effect of the validation, fire up your browser and give the URL for the genre management:

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

Then, click on the **New genre** link. In the next page, without entering any data, click on the **Create** button. If you get the following screen, then the validation has taken effect.



New genre

1 error prohibited this genre from being saved

There were problems with the following fields:

- Genre name can't be blank

Genre name

Description

Create

Back

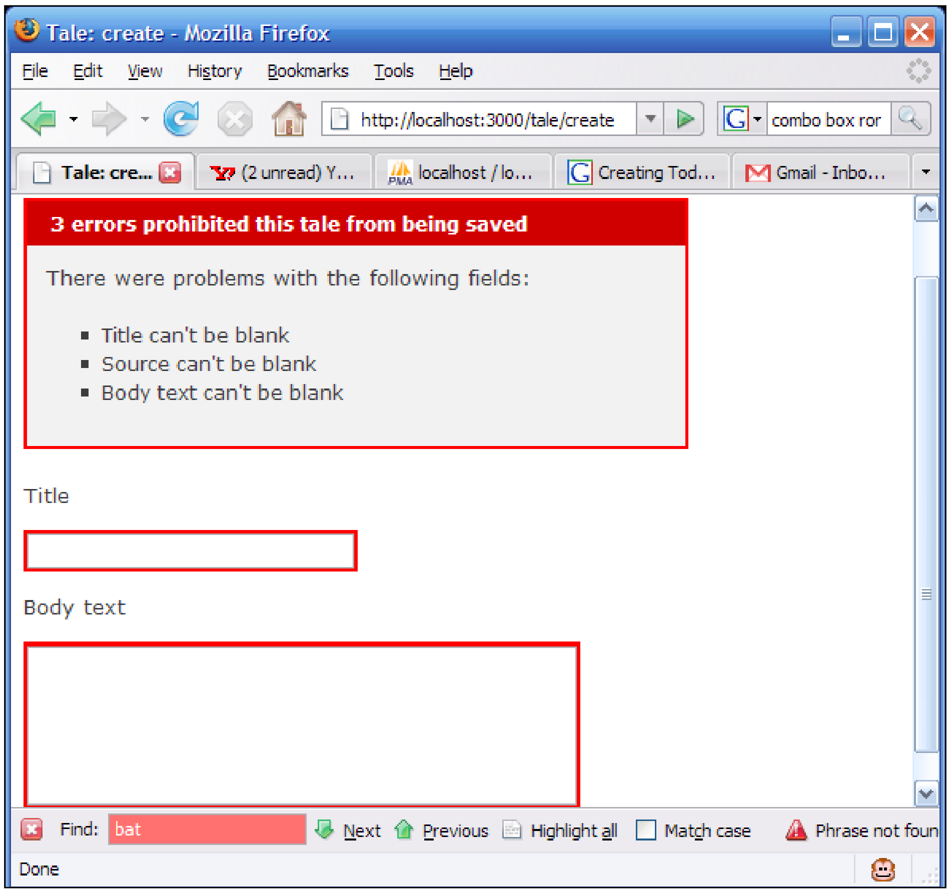
Next, let us apply required field validation on the `Tale` class. In the `Tale` class, we have multiple fields to validate. They are `title`, `body_text`, and `source`. The beauty of the `validate_presence_of ()` method is that you can give it multiple parameters, and thus you don't have to call it again and again. Once the validation has been added, the `Tale` class will be as follows:

```
class Tale < ActiveRecord::Base
validates_presence_of :title, :body_text, :source
belongs_to:genre
end
```

Now in the address bar, give the URL of the tale management module, which is:

`http://localhost:3000/tale`

Then click on the **New Tale** link and submit the form on the next screen without filling any values. You will get the following screen:


Tale: create - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/tale/create
combo box ror
Tale: cre... (2 unread) Y... localhost / lo... Creating Tod... Gmail - Inbo...
Find: bat Next Previous Highlight all Match case Phrase not foun
Done

3 errors prohibited this tale from being saved

There were problems with the following fields:

- Title can't be blank
- Source can't be blank
- Body text can't be blank

Title

Body text

That completes the customization of the generated Model as per our requirements. However, the relationship mapping has not been reflected on the Controller. That's what we are going to do next by customizing the generated Controller components.

## Customizing the Controller

We have made the Model components aware of the relationship, but the Controller and View are still not aware of it. If you look at the 'New Tale' form, you will see that there is no field to enter the genre type. To add the field we have to first make some changes in the Controller, specifically the `new` and `create` methods in the `tales_controller.rb` file. At present, the `new` method has the following statements:

```
def new
   @tale = Tale.new
end
```

It creates an instance variable named `tale` that is an object of the `Tale` class. To add genre to the 'New Tale' page, we have to retrieve the genres stored in the database. For that we will add a `find` method. As the `Tale` class has been derived from the Base class of the Active Record package, the `find` method and all its variations have been inherited by the `Tale` class. So we can call the `find_all` method on the object of the `Tale` class. The `find_all` method returns a list of the objects retrieved. In addition, we will be setting the value of the `status` attribute as it is not to be entered by the user. Therefore, the `new` method after adding the required statements will be:

```
def new
    @genres=Genre.find_all
    @tale = Tale.new
    @tale.status= “new"
end
```

Now, the obvious question will be why are we using the `new` method and not the `create` method? The `create` method is called after the data has been submitted. Also, the view corresponding to the `'new'` action is the 'New Tale' page.

Next change to be done is in the `create` method. The selected genre has to be added to the model. For that we have to set the value of `genre_id` attribute of the `Tale` object. So after adding the attribute the `create` method will look like:

```
def create
    @tale = Tale.new(params[:tale])
    @tale.genre_id=params[:genre])
    if @tale.save
      flash[:notice] = 'Tale was successfully created.'
```

```
        redirect_to :action => 'list'
      else
        render :action => 'new'
      end
    end
```

One point you have to keep in mind is that even if the form is built properly, checking the return value of the `save` method is always good. The reason is that while saving, if the database gets disconnected, the data will not be saved. By checking the return value of the `save` method, you can alert the user. Here, there are some new constructs being used, which are:

- **params**: It is an hash like structure containing the values of the form being submitted. The name of the fields act as index. So `params[:genre]` will return the value of the selected genre.
- **redirect_to**: This method redirects the user to the action or the view specified. Here `redirect_to :action => 'list'` redirects the user to the `list` action.
- **render**: It is used to output some data to the screen. It can take either an action or the text to be rendered as parameters. Here, we are using `action` as the parameter.
- **flash**: It is a temporary hash stored in the session. It is used to transfer the information between actions.

Now let us set the value of the story to `updated`, when the user edits the story. In the `edit` method of the `tales_controller.rb`, you will have to set the `status` attribute of the retrieved `Tale` object. After the addition, the code will look as follows:

```
def edit
    @tale = Tale.find(params[:id])
    @tale.status="updated"
end
```

The highlighted code contains the statement that has been added. That completes the changes to be made to the Controller component. Next, we will be looking at how it affects the View component.

[ An instance variable created in an `action` method is always accessible in the corresponding view. ]

## Refining the View

We have retrieved the details about genre from the database. Next, we have to show it to the user. However, changes need to be done to the View. The changes are:

- Refining the Add Tale template
- Refining the Edit Tale template

Let the process of refining begin.

### Refining the New Tale Template

Navigate to `app/views` folder. The folder will be having the following folders:

- **genres**: It contains the View templates for the genre corresponding to the `action` methods defined in `genre_controller.rb`.
- **layouts**: It contains the layout templates for the whole application. A file name `standard.rhtml` can be placed here that could contain layout which can be applied to the whole application. If you want to provide a different layout for a particular view, then you will have to provide the layout with the view name in this folder. For example, if you want to provide a different layout for genre, then you will have to create a `genre.rhtml` file in this folder. The scaffold specific layouts are also created here.
- **tales**: View templates for the tale corresponding to the action methods defined in tale_controller.rb.

The change that we will be making will be in tale's 'Add New Tale' template. So go into the `tales` folder. Open the `new.rhtml` file. You will find the following code.

```
<h1>New tale</h1>

<% form_tag :action => 'create' do %>
  <%= render :partial => 'form' %>
  <%= submit_tag "Create" %>
<% end %>

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

All the functions used here are form helpers. Let us look at each of these tags as they are used heavily.

- **form_tag**: It creates an HTML form tag. The `:action` parameter is (part of) the url hash that is used in the `action` attribute to post the info to. In reality, the `form_tag` takes a hash as parameter.

- **render**: It renders the fragment of page on the browser. The fragment can be extracted from another template using the hash named `:partial`. The `:partial` hash takes the name of the template from which the fragment has to be extracted. Here the name of the template is form.
- **submit_tag**: It creates a submit button. The name to be displayed is given as the parameter.
- **link_to**: It creates a link to a given action or a page. The parameters are the name to be displayed and the action or page to be called.

The helper that we have to consider here is `render`. Here it calls the `form` template. The convention that RoR follows in naming a template containing a fragment is prefixed with underscore (_). So the template that we need to change is `_form.rhtml`. Open `_form.rhtml`. The following is the code that it contains:

```
<%= error_messages_for 'tale' %>

<!--[form:tale]-->
<p><label for="tale_title">Title</label><br/>
<%= text_field 'tale', 'title'  %></p>

<p><label for="tale_body_text">Body text</label><br/>
<%= text_area 'tale', 'body_text'  %></p>

<p><label for="tale_submission_date">Submission date</label><br/>
<%= date_select 'tale', 'submission_date'  %></p>

<p><label for="tale_source">Source</label><br/>
<%= text_field 'tale', 'source'  %></p>

<p><label for="tale_status">Status</label><br/>
<%=@tale.status></p>
<!--[eoform:tale]-->
```

All the fields are created using the `form` helpers. Now, we have to add the `select` tag so that the user can select the `genre`. To do that we will iterate over the genres list and display the `select` tag. However, we will not be using the `form` helper. To the existing code, add the following highlighted statements. Also, the `status` field is deleted as the user need not enter the status.

```
<%= error_messages_for 'tale' %>

<!--[form:tale]-->
<p><label for="tale_title">Title</label><br/>
<%= text_field 'tale', 'title'  %></p>

<p><label for="tale_body_text">Body text</label><br/>
<%= text_area 'tale', 'body_text'  %></p>

<p><label for="tale_submission_date">Submission date</label><br/>
```

```
<%= date_select 'tale', 'submission_date'  %></p>

<p><label for="tale_source">Source</label><br/>
<%= text_field 'tale', 'source'  %></p>

<p><label for="tale_genre_name">Genre<br/>
<select name="genre">

<% @genres.each do |genre| %>
<option value="<%= genre.id %>">
<%= genre.genre_name %>

</option>

<% end %>
</select>
</p>
<!--[eoform:tale]-->
```

We are using iterator and block to iterate over the list of genres. The id of the genre is set as the value of the option and the name of the genre is set as the value to be displayed. There is a RoR helper to do the same. However, knowing an alternative method such as the one I described just now comes handy in circumstances where the helper cannot be used.

Next, let us change the view corresponding to the update of the tale.

## Refining the Edit Tale Template

Open the edit.rhtml file in the tale folder. It contains the following code:

```
<h1>Editing tale</h1>

<% form_tag :action => 'update', :id => @tale do %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Edit' %>
<% end %>

<%= link_to 'Show', :action => 'show', :id => @tale %> |
<%= link_to 'Back', :action => 'list' %>
```

As it is also calling `_form.rhtml`, we will have to change the fragment being called. The reason is that `_form.rhtml` contains code to create a select box with type of genres in it which we don't need while editing. So let's create a new template for showing the edit form. The name will be `_form_edit`. It will contain the following code:

```
<%= error_messages_for 'tale' %>

<!--[form:tale]-->
```

```
<p><label for="tale_title">Title</label><br/>
<%= text_field 'tale', 'title'  %></p>

<p><label for="tale_body_text">Body text</label><br/>
<%= text_area 'tale', 'body_text'  %></p>

<p><label for="tale_submission_date">Submission date</label><br/>
<%= date_select 'tale', 'submission_date'  %></p>

<p><label for="tale_source">Source</label><br/>
<%= text_field 'tale', 'source'  %></p>

<p><label for="tale_status">Status</label><br/>
<%= @tale.status  %></p>

<p><label for="genre_name">Genre</label><br/>
<%=@tale.genre.genre_name %></p>

<!--[eoform:tale]-->
```

As the `status` field won't be editable, we are not providing the text box for it. Similarly, there is no `edit` option for type of genre. Now, let us change the reference to `form_edit`. The `edit.rhtml` file will now look like:

```
<h1>Editing tale</h1>
<% form_tag :action => 'update', :id => @tale do %>
  <%= render :partial => 'form_edit' %>
  <%= submit_tag 'Edit' %>
<% end %>

<%= link_to 'Show', :action => 'show', :id => @tale %> |
<%= link_to 'Back', :action => 'list' %>
```

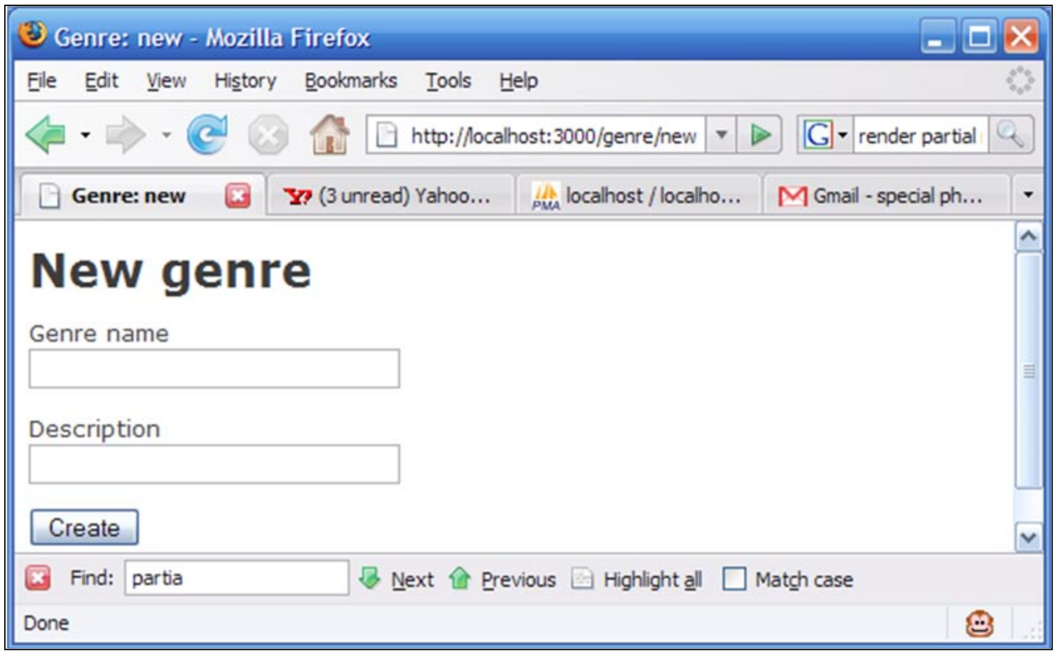
That completes the change to the updating functionality. Now, let us test the application.

## Testing the Application

The first thing to be done is add a genre. So open the browser with the following URL:

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

Now, click on the **New genre** link. You will see the following page:

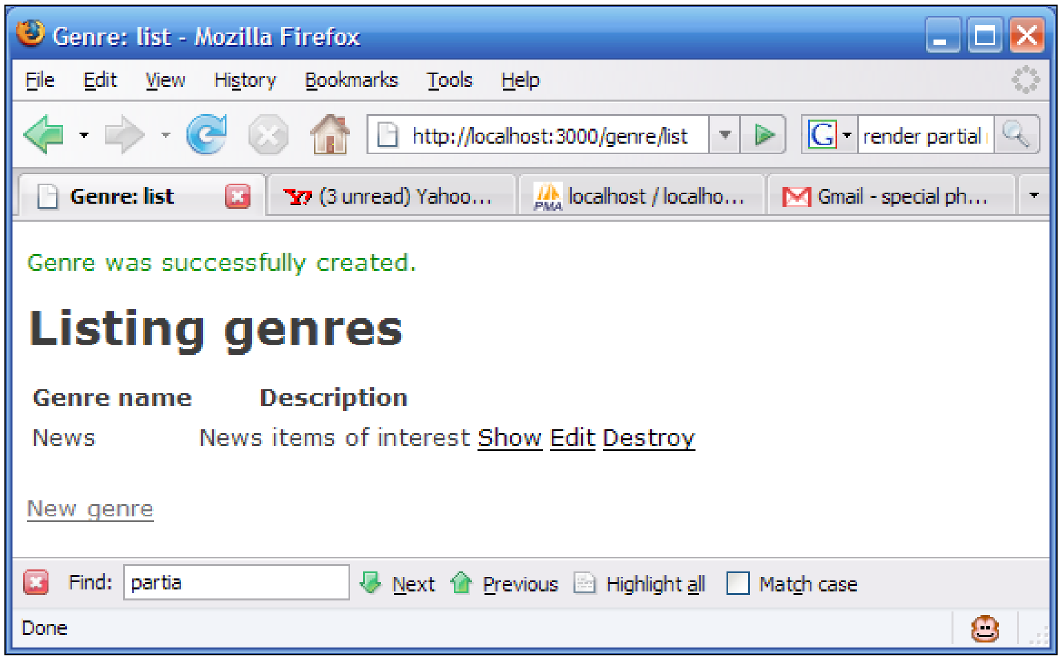


Add the following data:

**Genre name**—News

**Description**—News items of interest

Now, click on the **submit** button. If you get the following page, then everything is fine.



Genre: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/genre/list
render partial
Genre: list
(3 unread) Yahoo…
localhost / localho…
Gmail - special ph…

Genre was successfully created.

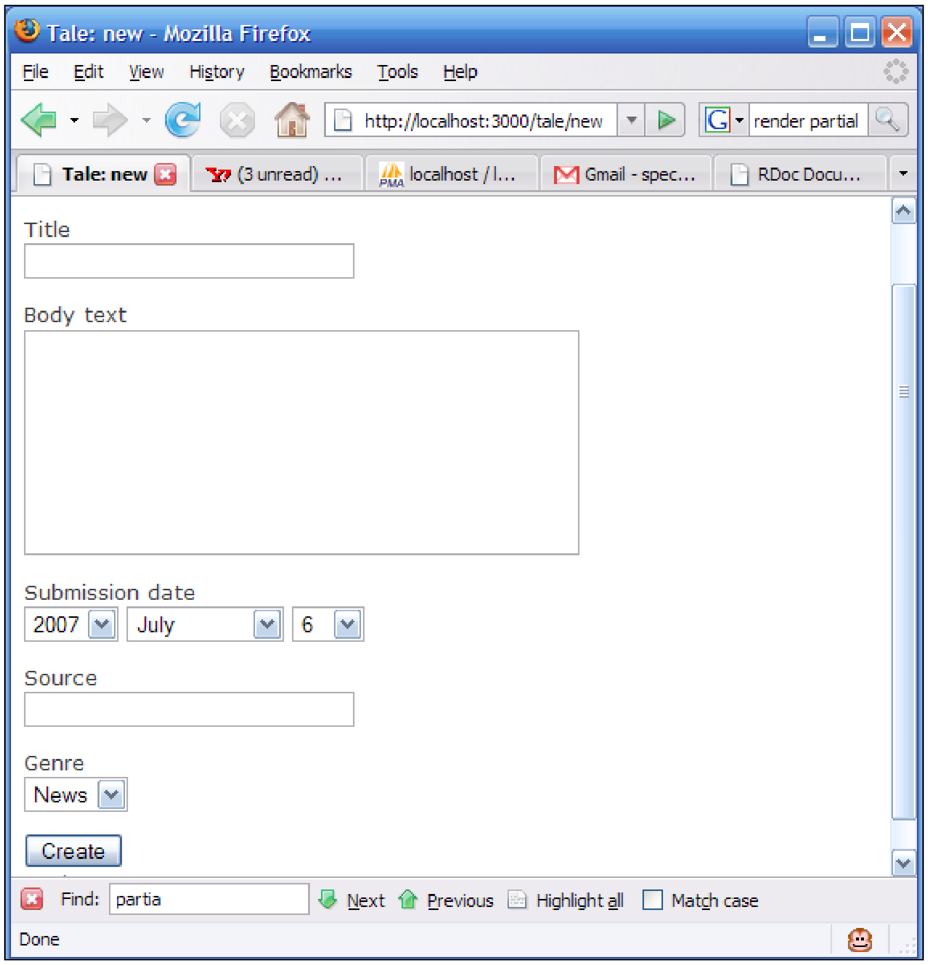
Listing genres

| Genre name | Description | | | |
|---|---|---|---|---|
| News | News items of interest | Show | Edit | Destroy |

New genre

Find: partia
Next Previous Highlight all Match case
Done

Now open the URL `http://localhost:3000/tale` in the browser and click on the **New Tale** link. You will get the following page:



Title

Body text

Submission date
2007 July 6

Source

Genre
News

Create

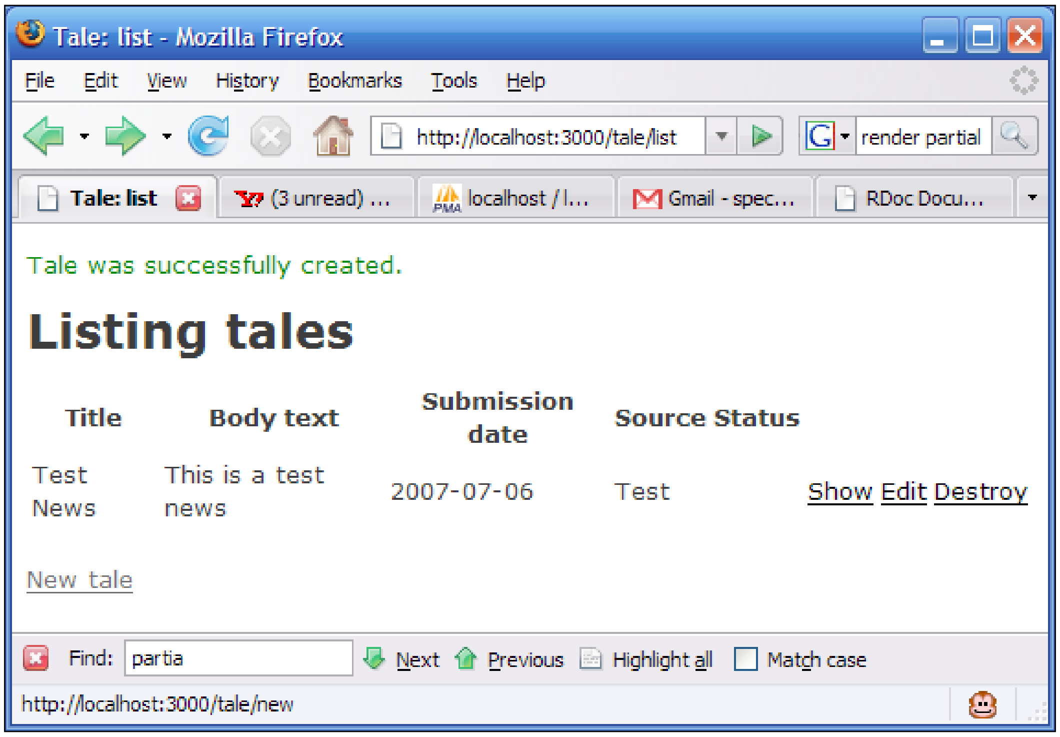
Add the following data:

**Title – Test News**

**Body text – This is a test news**

**Source Status – Test**

For the other fields, leave the default values. On clicking the **Create** button you will get the following page:


Tale: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/tale/list
render partial
Tale: list
(3 unread) ...
localhost / l...
Gmail - spec...
RDoc Docu...
Find: partia
Next Previous Highlight all Match case
http://localhost:3000/tale/new

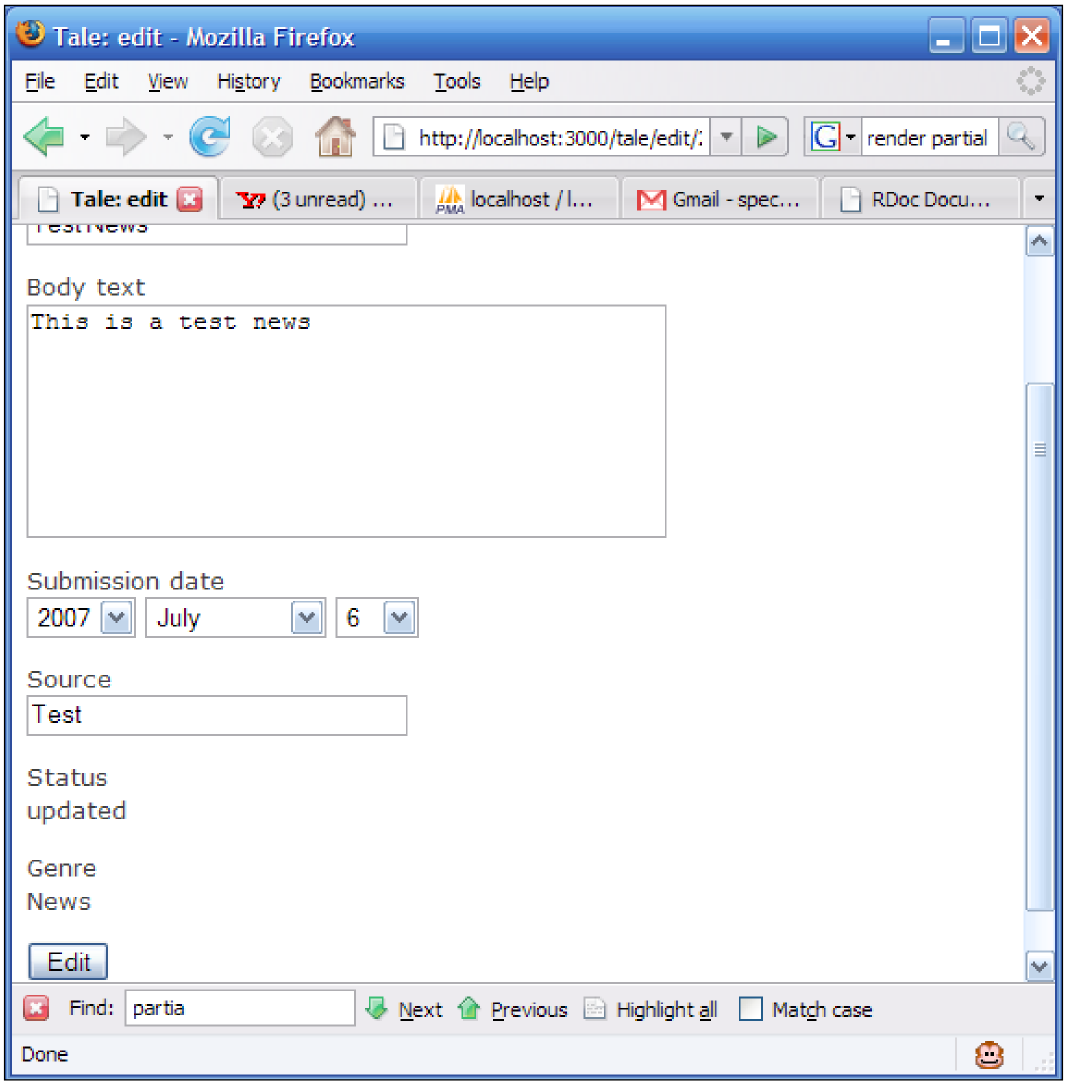
Tale was successfully created.

Listing tales

| Title | Body text | Submission date | Source Status | |
|---|---|---|---|---|
| Test News | This is a test news | 2007-07-06 | Test | Show Edit Destroy |

New tale

Next, click on the **Edit** link. You will get the following screen:



Body text
This is a test news

Submission date
2007 July 6

Source
Test

Status
updated

Genre
News

Change the value of the Body text to “This was test news" and click **Edit**. It will show you the list with the edited text. That confirms the success of the changes we have done. That completes the testing of our application.

## Summary

That completes another chapter. In this chapter, you understood how to design tables according to the conventions of RoR, creation of scaffolds for tables, and changing the scaffolds according to the requirements.

This is just the start of our application. Keep on reading.

# 4 Managing the Users

In the last chapter, we developed one of the core modules of TaleWiki. In this chapter, we will be dealing with the next core module—the User Management module. You would have reached a conclusion by now that the Tales without the corresponding users lacks credibility. To ensure credibility of the stories submitted, as well as to track the users using the system, User management is required.

This chapter will start with a detailed look at the requirements of the User Management module. Then we will design the module according to the requirements and develop the module functionalities. The chapter will end with testing the functionality of the developed module. So let's get started.

## Understanding the Requirements

User Management is a module that is required by almost all the applications. However, the functionalities provided by it will differ from application to application. This is where the module specific functionalities come into the picture. As TaleWiki is a collaborative site that will be built upon story submissions, the different users will have different privileges so that the matter submitted can be verified and edited when required. Also, a separate user will be there, who will be looking after the site. That user is the administrator. So, we can say that the functionalities to be provided by the User Management can be broadly divided into two:

- Managing the User
- Managing the Privileges

The former will be how to go about managing the users and the latter will be about who can access and what they can modify. Let us have a detailed look into these.

## Managing the User

After stories, users form the most important aspect of our system, as all other functionalities will be based on tales and users. As stated in the previous chapter, each story is submitted by a user. So it becomes necessary that we keep track of who submits what. We dealt with the management of tales in the previous chapter. Now let us look at what requirements arise when we introduce users into the picture.
To keep track of which user submitted the story, we will need to develop the following functionalities:

- **Registering the User**: It's the first step in managing the users. It will let the administrator know who is currently using the system. The user registration can also be termed as adding the user if we look at it from the perspective of the administrator. These terms will be used interchangeably in this chapter. When a user registers himself or herself with TaleWiki, the information collected will include the desired user id, full name of the user, age, complete address, email id, and gender. The age will be required so that we can assure that no one under the age of 15 gets registered.
- **Assigning Role**: Role translates to privilege. Each user will be given a role so that whatever functionalities of TaleWiki that they can access can be clearly defined. This can be achieved by assigning roles to the registered users. Until the roles are assigned, the users will be treated as 'Guest.' A user will not have more than one Role.
- **Modifying the Information about the Users**: The user as well as the administrator (on being requested by the user) may want to modify some information provided at the time of registration. This requirement will be covered by 'Modifying User Details' functionality. All the information except *user id* can be changed using this functionality. However, the normal user and the administrator will have different views. The normal user will be able to see and change his or her details. But the administrator will be able to view and change all the details of all the users.
- **Viewing the Users**: The administrator will need to view all the registered users and their details, including the stories submitted by them. This sub-module will provide the administrator with two views—*list* view and *detailed* view. The *detailed* view will contain the details of the stories submitted by the particular user.
- **Deleting a User**: There may be a case where the user may need to register off the system and his/her details will need to be deleted. In such situations, *delete* functionality will delete a user and the user's associated details, including the submitted stories.

That completes the round up of functionalities to be provided by the User Management sub-module. Next, let us look at the Role Management sub-module.

## Managing Roles

We have a system that will be used by multiple users. So firstly, there is a need to stop users from accessing certain functionalities. This is done by a process known as 'Authorization.' For authorization to work, we need to specify the privileges for each user. Through privileges, we can check what functionalities of TaleWiki can be used by a particular user. Checking privileges for each user is a time-consuming job. Hence, privileges are grouped together as Roles. Each Role represents a set of privileges. Thus, when you assign a Role to a user, the set of privileges that the Role represents is also assigned to the user.

The privileges that you can assign to a Role can be either static or dynamic. If privileges are static then once the Roles have been defined, the privileges cannot be changed. That means you will not be able to add new privileges or delete/modify the existing ones from a Role. But in the case of dynamic privileges, you can add new privileges to the existing set corresponding to a Role. In the case of TaleWiki, we will be using static privileges. This is to ease the Role management until the interface for the administrator is developed. After ascertaining the type, privilege assignment is decided. The next step is to decide the functionalities needed to manage the Roles. Here are the functionalities that the Role Management module will be providing to the administrator:

- **Add Role**: To assign a Role, the system should know what Roles are supported. 'Add Role' functionality will help the administrator to add new Roles so that TaleWiki can know about the Roles that can be assigned. The information that can be entered will include the *id* of the Role and the *name* of the Role.
- **Modify Role**: In case the administrator wants to change the name associated with a Role, he/she will be able do it using this functionality.
- **Delete Role**: There may be situations where a particular Role may no longer be required. When faced with such situations, the administrator can delete that Role. However, the Role won't be deleted as long as users are still assigned to that Role.
- **View Role**: This functionality will help the administrator in viewing the Roles supported by TaleWiki. The *list* view will contain a list of all the supported Roles, and the *detailed* view will contain the details of a particular Role, including the users who are assigned that Role.

That's all about the functionalities that will be provided by the User Management module. The next task-at-hand is to design the tables that will become the back-end of the user management functionality.

## Designing the Tables

As you remember from the previous chapter, the next step is to design the tables. To create tables, we need to understand the entities and their relationship, the schema corresponding to the entities, and then the table creation queries. If we go step-by-step, we can say that following are the steps in designing the tables for the User Management module:

- Designing the E-R model
- Deriving the Schema from the E-R model
- Creating the Tables from the Schema

So, let us follow the steps.

### Designing the E-R Model

To design the E-R model, let us first look at what we have understood about the data required by the functionalities, which we just discussed. It tells us that 'only the Users with a particular Role can access TaleWiki'. Now we can consider this as our 'problem statement' for our E-R model design. If you observe closely, the statement is vague. It doesn't tell about the particular Roles. However, for the E-R design, this will suffice as it clearly mentions the two main entities, if we use the E-R terminology. They are:

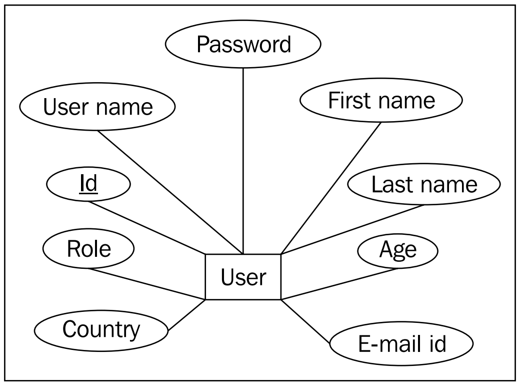
- User
- Role

Let us look at the User entity. Now this entity represents a real-world user. It is not difficult to describe its attributes. Keeping a real-world user in mind and the functionalities discussed for managing a user, we can say that the User entity should have the following attributes:

- **Id**—It will identify the different users, and it will be unique.
- **User name**—the name which will be displayed with the submitted story.
- **Password**—the pass key with which the user will be authenticated.
- **First name**—the first name of the user.

- **Last name**—the last name of the user. The combination of the first and last name will be the real name of the user.
- **Age**—the age of the user. This will help in deciding whether or not the user is of required age which is 15.
- **E-mail id**—the email id of the user in which he/she would like to get emails from the administrator regarding the submissions.
- **Country**—to keep track of the 'geographic distribution' of users.
- **Role**—to know what privileges are granted for the user. The Role is required because the problem statement mentions "User with a particular Role".

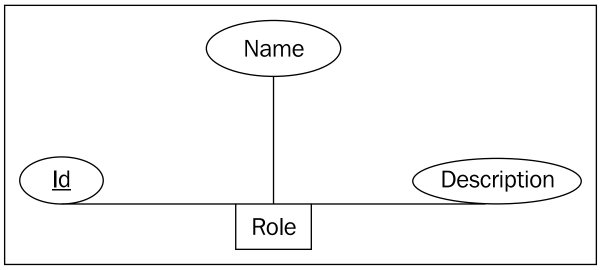
The entity diagram will be as follows:



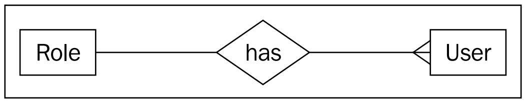
Next, let us look at the Role entity. Role, as already discussed, will represent the privileges a user can have. And as these privileges are static, the Role entity won't need to have the attribute to store the privileges. The important point about the static privileges that you have to keep in mind is that they will have to be programmatically checked against a user. In other words, the privileges are not present in the database and there can only be a small number of Roles with predefined privileges. We will be discussing more about it while developing the interface for managing Roles. Keeping this in mind, we can say that the Role entity will have the following attributes:

- **Id**—the unique identification number for the Role.
- **Name**—the name with which the id will be known and that will be displayed along with the user name.

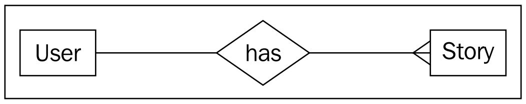
The entity diagram for Role entity will be as follows:



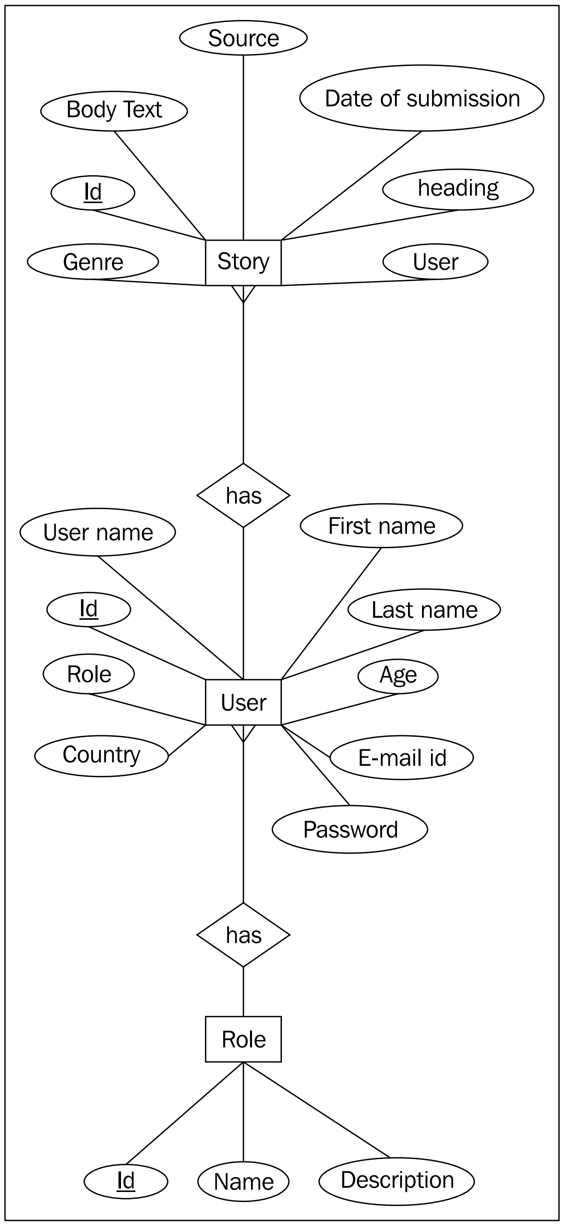
We have completed two out of three steps in designing the E-R model. Next, we have to define how the User entity is related with the Role entity. From the problem statement we can say that a user will definitely have a Role. And the functionality for assigning the Role tells us that a user can have only one Role. So if we combine these two, we can say that 'A user will have only one Role but different users can have the same Role'. In simple terms, a Role—let us say normal user—can be applied to different users such as John, or Jane. However, the users John or Jane cannot be both normal user as well as administrator. In technical terms, we can say that a Role has a one-to-many relationship with the User entity and a User has a many-to-one relationship with a Role. Diagrammatically, it will be as follows:



One piece of the puzzle is still left. If you remember the Story entity from the previous chapter, we had found that each story had a submitter. The submitter is a user. So that means there is a relationship between the User and the Story entity. Now, a user, let us say, John or Jane, can submit many stories. However, the same story cannot be submitted by more than one user. On the basis of this we can say that a User has a many-to-one relationship with a Story and a Story has a many-to-one relationship with a User. According to the E-R diagram it will be as follows:



The final E-R design including all the entities and the attributes will be as follows:



That completes our E-R design step. Next, we will derive the schema from the E-R model.

## Deriving the Schema

We have all we need to derive the schema for our purpose. While deriving a schema from an E-R model, it is always a good choice to start with the entities at the 'one' end of a 'one-to-many' relationship. In our case, it is the Role entity. As we did in the previous chapter, let us start by providing the details for each attribute of the Role entity. The following is the schema for the Role entity:

| **Attribute** | **Data type of the attribute** | **Length of the acceptable value** |
|---|---|---|
| Id | Integer | 10 |
| Name | Varchar | 25 |

Next, let us look at the schema of the User entity. As it is at the 'many' end of the 'one-to-many' relationship, the Role attribute will be replaced by the Id of Role. The schema will be as follows:

| **Attribute** | **Data type of the attribute** | **Length of the acceptable value** |
|---|---|---|
| Id | Integer | 10 |
| User name | Varchar | 50 |
| First name | Varchar | 50 |
| Last name | Varchar | 50 |
| Password | Varchar | 15 |
| Age | Integer | 3 |
| e-mail id | Varchar | 25 |
| Country | Varchar | 20 |
| Id of the Role | Integer | 10 |

Now, let us revisit the Story entity. The attributes of the entity were:

- Id—This is the Primary key attribute as it can uniquely identify a story.
- Heading—the title of the story.
- Body text—the body of the story.
- Date of Submission—the day the user submitted the story.
- Source—the source from where the story was found. If it is written by the user himself/herself, the source will be the user's id.
- Genre—the category of the story.
- User—the user who submitted the story.

Now based on the relationship that we have established between the User Entity and the Story entity, we can say that Story is at the 'many' end of the relationship. So, in its schema, the User attribute will be replaced by the Id of the User entity. Thus, the schema of Story will be:

| **Name of the attribute** | **Data type of the attribute** | **Length of the acceptable value** |
|---|---|---|
| Id | Integer | 10 |
| Title | Varchar | 100 |
| Body Text | Varchar | 1000 |
| Date of Submission | Date | |
| Source | Varchar | 50 |
| Status | Varchar | 15 |
| Id of Genre | Integer | 10 |
| Id of the User | Integer | 10 |

The schema has been derived and now we can move to the last part of the database design—creation of the tables.

## Creating the Tables

Looking at the schema and keeping in mind the conventions required for tables in RoR, here is the table creation statement for the Role schema:

```
CREATE TABLE `roles` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`name` VARCHAR( 25 ) NOT NULL ,
`description` VARCHAR( 100 ) NOT NULL
) ENGINE = innodb;
```

Next comes the table creation statement for the User schema. Note that here also we are following the one-to-many path, that is, the table at the 'one' end is created first. Whenever there is a one-to-many relationship between entities, you will have to create the table for the entity at the 'one' end. Otherwise you will not be able to create a foreign key reference in the table for the entity at the 'many' end, and if you try to create one, you will get an error (obviously). So here is the create table statement for the User schema:

```
CREATE TABLE `users` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`user_name` VARCHAR( 50 ) NOT NULL ,
`password` VARCHAR( 15 ) NOT NULL ,
`first_name` VARCHAR( 50 ) NOT NULL ,
`last_name` VARCHAR( 50 ) NOT NULL ,
`age` INT( 3 ) NOT NULL ,
`email` VARCHAR( 25 ) NOT NULL ,
`country` VARCHAR( 20 ) NOT NULL ,
`role_id` INT NOT NULL,
CONSTRAINT `fk_users_roles` FOREIGN KEY (`role_id`) REFERENCES `role`(
`id`) ON DELETE CASCADE
) ENGINE = innodb;
```

Next, let us alter the table for story, that is, the `tales` table, by adding a foreign key reference to the `users` table in it. For that, first we will have to add a new column name `user_id` in the `tales` table. Here is the query for adding a new column:

```
ALTER TABLE `tales` ADD `user_id` INT NOT NULL ;
```

Next, we will make a reference to the users table with the following query:

```
ALTER TABLE `tales` ADD FOREIGN KEY ( `user_id` ) REFERENCES `users` (
`id`
) ON DELETE CASCADE ;
```

That completes our task of creating the required tables and making necessary changes to the tales table. The effect of this change will be visible to you when we implement session management in the next chapter. And incidentally, it completes the 'designing the tables' section. Let us move onto the development of the user management functionality.

## Developing the User Management

The tasks and operations for user management are clear and the tables are ready. The only job left to be done is developing the module itself so that user will be able to access the functionalities. User Management has two different sets of functionalities – managing the roles and managing the users. So we will be developing these as two different set of functionalities. First we will be working on Role Management and then we will move over to User Management. The steps will be the same as in the previous chapter. However, unlike the previous chapter, we will not be focussing on the steps. Instead we will look at both functionalities separately and steps will be a part of them. This will be the approach from this chapter onwards because the steps have already been introduced in Chapter 2. So let us get started.

## Developing the Role Management

The steps in developing the interface for Role Management are the same as follows:

- Generating the Scaffolds
- Modifying the Model
- Customizing the Controller
- Refining the View

But in this case the major refinement or changes will be done to the View. The reason is the *detailed* view functionality.

### Generating the Scaffolds

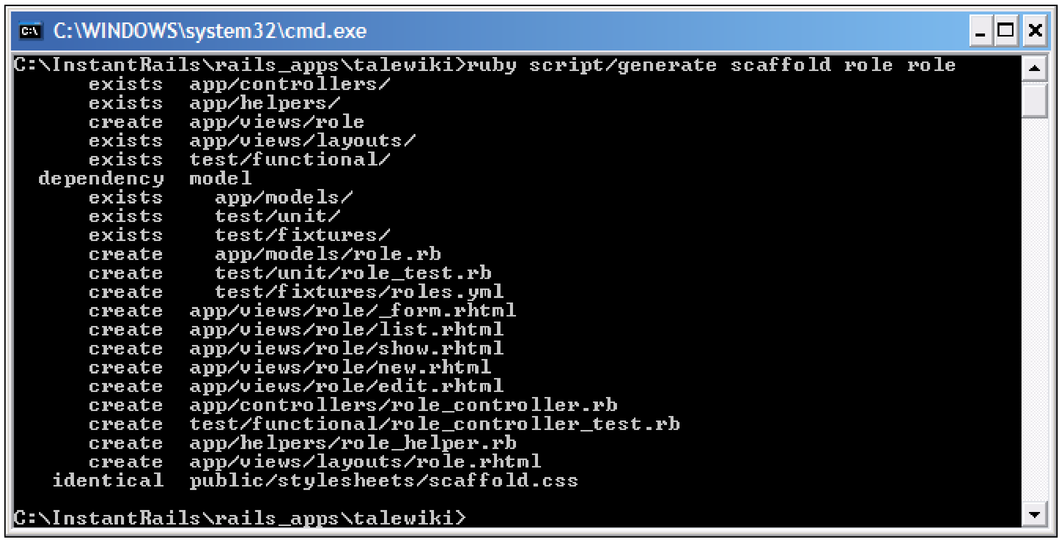
Give the following commands at the prompt to set the paths and get into our application directory:

```
use_ruby
cd talewiki
```

Once inside the folder of our application, give the following command to generate the scaffold:

```
ruby script/generate scaffold role role
```

In response, you will get the following screen:



```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold role role
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/role
      exists  app/views/layouts/
      exists  test/functional/
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/role.rb
      create    test/unit/role_test.rb
      create    test/fixtures/roles.yml
      create  app/views/role/_form.rhtml
      create  app/views/role/list.rhtml
      create  app/views/role/show.rhtml
      create  app/views/role/new.rhtml
      create  app/views/role/edit.rhtml
      create  app/controllers/role_controller.rb
      create  test/functional/role_controller_test.rb
      create  app/helpers/role_helper.rb
      create  app/views/layouts/role.rhtml
   identical  public/stylesheets/scaffold.css

C:\InstantRails\rails_apps\talewiki>
```

Next, we have to ensure that the name field of the Role in the 'New Role' interface is never left empty by the user. For that, we will now modify the Model.

## Modifying the Model

In the case of Role Management, the only change we will be doing in the Model is adding the validation for the name attribute. We do not want anyone to add a Role without a name, do we? To avoid that, open the `role.rb` file from the `app/models` folder. It contains the following code:

```
class Role < ActiveRecord::Base
end
```

It tells RoR that the `Role` class is mapped to the roles table (through convention). Next, let us add the `validates_presence_of` method with `:name` as the argument to validate the name field. Also, let us check the uniqueness of the name of the Role, as it is important that the name is unique.

```
class Role < ActiveRecord::Base
   validates_presence_of :name
   validates_uniqueness_of :name

end
```

With that, the modifications to the data model of the Role Management are completed. Next, we have to add the details of users assigned to a particular Role. For that, first we have to tell the model that it will be containing objects of the User model. So, add the `has_many` declaration to the `role.rb`, shown as follows:

```
class Role < ActiveRecord::Base
      validates_presence_of :name
      validates_uniqueness_of :name
      has_many :users
end
```

We will be changing the Controller and the View. So let's move to the changes to be done in the Controller. However, keep in mind that we will not be testing the Role Management until we create the User model and tell it that the User model is related with the Role model. At present, there is no User model. We are creating the placeholders so that when we create the User model we don't need to revisit the Model corresponding to the Role.

## Customizing the Controller

To show the details of the users assigned, we need to get the list of users having the Role. In order to display more information about the user in the details page, we will have to make changes in the action corresponding to the details page. By convention, it is named `show`. The `show` action method for the Role Controller has the following code:

```
def show
    @role = Role.find(params[:id])
  end
```

You will find this method in `role_controller.rb` within the `app/controllers` folder. We have to show the user under the particular role. For that, get the list of users from the `role` object by calling users on the `role` object. The code will be as follows:

```
def show
    @role = Role.find(params[:id])
    @users=@role.users
  end
```

Now may be wondering where the `users` attribute came from and what it actually gives you as data. The answer is that the `has_many` declaration in Model tells RoR that we are expecting many instances of the User object corresponding to a Role object. The best way to represent many objects within another object is by using a list. That's what RoR is doing here. So when you create an object of Role using any form of the `find` method, RoR also looks at the corresponding User objects and makes a list populated with them. Then it creates a getter for the created list. The name of the getter is again a convention. So it is called users because it is plural of the user class. That completes the explanation behind the `@role.users` expression. Now you can understand how much work RoR does for us and the level of abstraction it provides to make our development easy. Next, let us do the required changes in the View.

## Refining the View

The next step in showing the details is to modify the template corresponding to the `show` action method. In this case it is the `show.rhtml` file within the `app/views/role` folder. It will has the following code:

```
<% for column in Role.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @role.send(column.name) %>
</p>
<% end %>
<%= link_to 'Edit', :action => 'edit', :id => @role %> |
<%= link_to 'Back', :action => 'list' %>
```

The code iterates over the attributes of the selected object and displays their human readable format. What we will be doing is adding the code to iterate over the list of users and displaying their details. This is how you do it:

```
<% for column in Role.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @role.send(column.name) %>
</p>
<% end %>

<table>
<tr>
       <th>UserName</th>
       <th>FirstName</th>
       <th>LastName</th>
       <th>email id</th>
</tr>

<% for user in @users %>
  <tr>
      <td><%=user.user_name%></td>
      <td><%=user.first_name%></td>
      <td><%=user.last_name%></td>
      <td><%=user.email%></td>
  </tr>
<%end%>
</table>

<%= link_to 'Edit', :action => 'edit', :id => @role %> |
<%= link_to 'Back', :action => 'list' %>
```

The highlighted code is actually iterating over the list of users by using the for construct. Then the attributes of the current user object is being displayed using normal HTML and embedded Ruby code. That completes one end of the User management. But it is still not complete as the functionality to manage users is still not in place. So let us put User management in place next.

## Developing the User Management Functionality

We just completed the first half of the User Management module by developing the Role management functionality. Without further ado, let us get on to the second half of User management. To develop the interfaces for the User management functionality, the steps are the same as that we followed for Role management, which are:

- Generating the Scaffold
- Modifying the Model
- Providing Default Role to the User
- Adding Display Action method to the Controller
- Refining the View
- Adding the Assign Action Method to the Controller

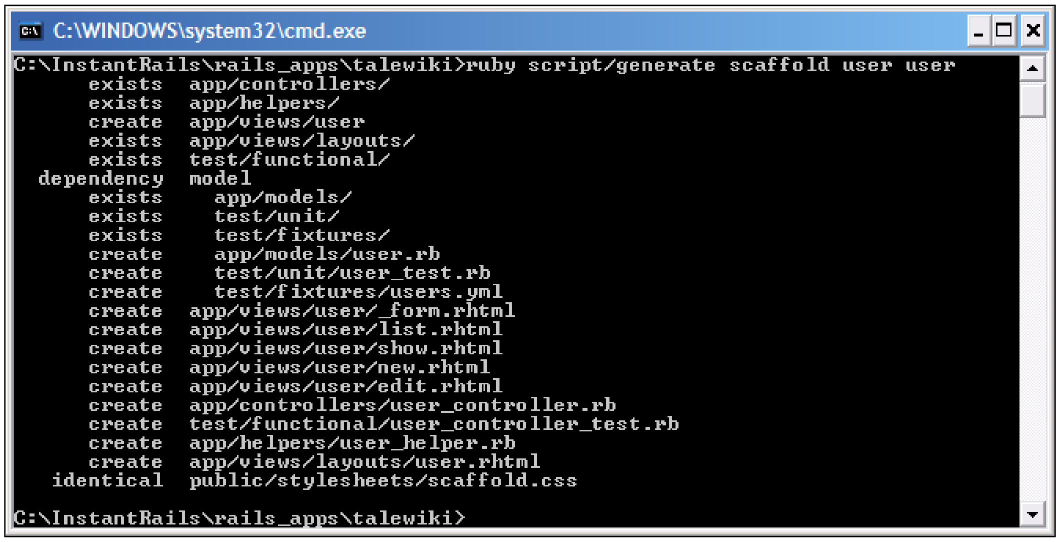
As in Role management, there is only one operation that we will be customizing—assigning a Role to a User. As RoR doesn't generate the scaffold for the assigning role, the better term will be adding an operation. So here is how we are going to do it.

## Generating the Scaffold

At the command prompt, give the following command to generate the scaffold for the `users` table:

```
ruby script/generate scaffold user user
```

Just like before, you will get the following screen:



C:\WINDOWS\system32\cmd.exe

```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold user user
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/user
      exists  app/views/layouts/
      exists  test/functional/
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/user.rb
      create    test/unit/user_test.rb
      create    test/fixtures/users.yml
      create  app/views/user/_form.rhtml
      create  app/views/user/list.rhtml
      create  app/views/user/show.rhtml
      create  app/views/user/new.rhtml
      create  app/views/user/edit.rhtml
      create  app/controllers/user_controller.rb
      create  test/functional/user_controller_test.rb
      create  app/helpers/user_helper.rb
      create  app/views/layouts/user.rhtml
   identical  public/stylesheets/scaffold.css

C:\InstantRails\rails_apps\talewiki>
```

The setup for User management is completed. Now, let us move onto customization. The first part of customization is validating the data input and telling RoR that the `users` table is related to the `roles` table. That's what we are going to do next.

## Modifying the Model

We have two tasks related to the model—adding the validation and relating the user model with the model of Role. It is always better to do the validation first so that during the saving of data overhead can be reduced. The validations we will be doing are:

- The field's user name, password, first name, last name, age, email, and country should be filled. For this we will use the `validates_presence_of ()` function.
- The user name has to be unique. We will be using the `validates_uniqueness_of ()` method to check the uniqueness of the user name.
- Email should be of the form `some@some.com`. To check the format of email, we will use the `validates_format_of ()` function. Apart from the field to be validated, it expects a regular expression which will be used to validate the value of the field.
- The age should be of numerical value. The `validate_numericality_of ()` function will take care of the validation for numerical value in age field.

To add these validations, open the `user.rb` file from the `app/models` folder and add the validation code to the `User` class. The class currently is as follows:

```
class User < ActiveRecord::Base
end
```

After adding the code, the class would be as below:

```
class User < ActiveRecord::Base
   validates_presence_of :user_name, :password, :first_name,
   :last_name, :age, :email, :country
   validates_uniqueness_of :user_name
   validates_numericality_of :age
   validates_format_of :email, :with => /\A([^@\s]+)@((?:[-a-
                                         z0-9]+\.)+[a-z]{2,})\Z/i
end
```

The regular expression is passed with the help of the `:with` hash. That completes the validation aspect of the User management. Next, we have to tell RoR that the `users` table is related to the `roles` table. We will do that using the `belongs_to` declaration, as the `users` table is at the many end of the relationship. Now, add the `belongs_to` declaration after the validation code. The complete class definition will be as follows:

```
class User < ActiveRecord::Base
   validates_presence_of :user_name, :password,:first_name,\
                         :last_name, :age, :email, :country

  validates_uniqueness_of :user_name
  validates_numericality_of :age
 validates_format_of :email,:with =>/\A([^@\s]+)@((?:[-a-z0-\
                        9]+\.)+[a-z]{2,})\Z/i

  belongs_to :role

end
```

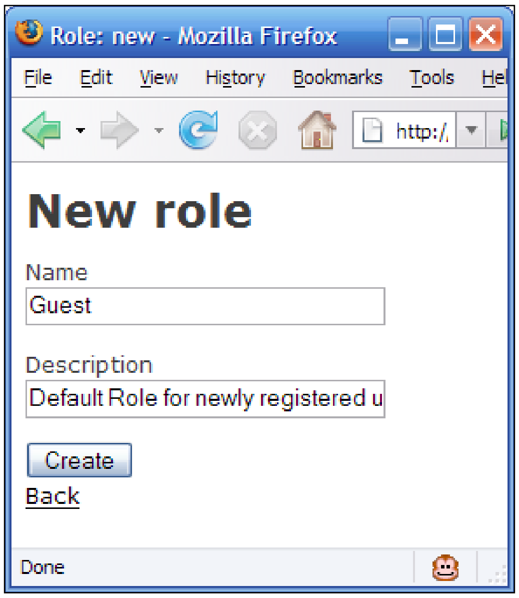
Next, we have to assign the user a default role which will be 'guest'. Let us see how to do it.

## Assigning Default Role to a User

To assign a default Role, first let us create a Role named 'Guest'. Navigate to the 'New Role' page after giving the following URL in the address bar of the browser:

`http://localhost:3000/role`

For the **Name** field supply **Guest** as the value. For the **Description** field, give **Default Role for newly registered user** as the value. The page will look as follows:


Role: new - Mozilla Firefox
File Edit View History Bookmarks Tools Hel
http:/
Create
Back
Done

New role

Name
Guest

Description
Default Role for newly registered u

Now, click **Create**. The 'Guest' Role is now created. So let us now assign it to any user who is registering. For that go to the `create` method in the `UserController` class defined in the `user_controller.rb` file within the `app/controllers` folder. Add these two statements to the `create` method before the `if` statement:

```
@role=Role.find(:all, :conditions=>"name='Guest'")
@user.role_id=@role.id
```

What the first statement is doing is that it is retrieving the objects of `Role` having their name as `Guest`. The second statement assigns the `id` of the retrieved `Role` object (we are sure that there is only one Role object because of the uniqueness check within the model corresponding to the Role) to the `role_id` attribute of the `user` object. After addition of the statements, the `create` method will be as follows:

```
def create
    @user = User.new(params[:user])
    @role=Role.find(:all, :conditions=>"name='Guest'")
    @user.role_id=@role.id
    if @user.save
      flash[:notice] = 'User was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end
```

Next, let us add the `action` method to display all the users and all the roles so that the assign role functionality can be developed.

## Adding Display Action Method to the Controller

The two things that we require to assign a role to a user are the list of registered users and the available roles. So first we have to add an action method that will retrieve both the lists—that of users and roles. For retrieving the lists we will use the `find(:all)` function on both the models. Add the following method to the `UserController` class in the `user_controller.rb` file. You can find the file in the `app/controllers` folder.

```
def display_assign
   @users = User.find(:all)
   @roles = Role.find(:all)
end
```

So we have created the two required lists. Now let us display it to the user.

## Refining the View

The first step in displaying both the lists is to create a template in the app/views/ users folder. Let us name it display_assign.rhtml. It will contain code to show two combo boxes—one for showing the users and another for showing the roles. Apart from that we need to give the submit button to submit the selected values to the Controller. The code is identical to what you have seen for displaying the list of genres in the 'New Tale' page. The following is the code:

```
<h1>Assign Role</h1>
<% form_tag :action => 'assign' do %>
  <p><label for="user_user_name">User<br/>
<select name="user">
<% @users.each do |user| %>
<option value="<%= user.id %>">
<%= user.user_name %>
</option>
<% end %>
</select>
</p>
<p><label for="role_role_name">Role<br/>
<select name="role">
<% @roles.each do |role| %>
<option value="<%= role.id %>">
<%= role.name %>
</option>
<% end %>
</select>
</p>
  <%= submit_tag "Assign Role" %>
<% end %>
<%= link_to 'Back', :action => 'list' %>
```

The code creates two combo boxes and fills them with user names and roles. It then creates a submit button. All of this is enclosed within the form helper. The action method being called is assign. That completes the page to display users and roles so that a role can be assigned to a user. Now we have to save the data in table.

## Adding the Assign Method to the Controller

When the user clicks on **submit,** the selected user and role has to be saved. Essentially, this means the `role_id` of the selected user has to be updated with the `id` of the selected role. So the `assign` method in `UserController` will be as follows:

```
def assign
    @user= User.find(params[:id])

    if @user.upadate_attribute :role_id,params[:role]
      flash[:notice] = 'User was successfully assigned the role.'
      redirect_to :action => 'list'
    else
      flash[:notice] =
                 'Role could not be assigned to the selected user.'
      render :action=>'display_assign'
    end
  end
```

It updates only the selected attribute of the user object. If successful, it shows the list of users, otherwise it redisplays the 'Assign Role' page. In this case it is `role_id`. That completes our assign role task. Next, let us test the application.

# Testing the Functionalities

We are going to test only those functionalities that have been either modified or added by us after the scaffold has been generated. So the functionalities to be tested are:

- Uniqueness of the Role name
- Validation of the User details during user registration/addition
- Assigning of Role to a User
- Displaying of Users assigned a particular Role

Let us begin the testing with the uniqueness of the Role name. Open the following URL in a browser:

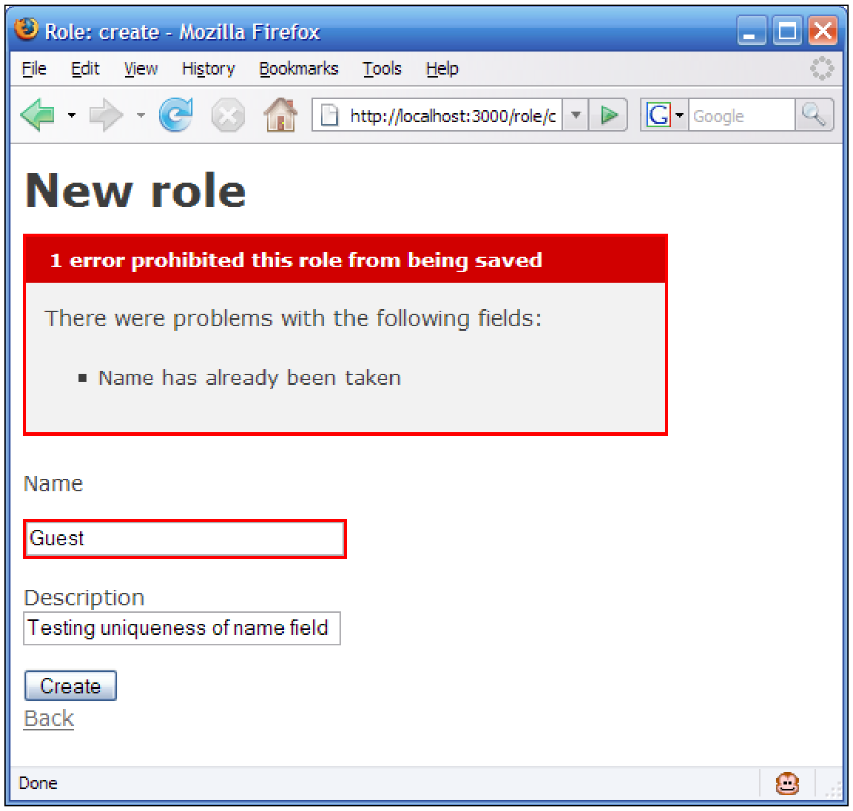
`http://localhost:3000/role`

Navigate to the **New role** page. In the page, give the following values to the fields:

**Name—Guest**

**Description—Testing uniqueness of name field**

Now, click on **Create**. If you get the following screen, then the validation is working.


Role: create - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/role/c
Google

New role

1 error prohibited this role from being saved

There were problems with the following fields:

- Name has already been taken

Name: Guest

Description: Testing uniqueness of name field

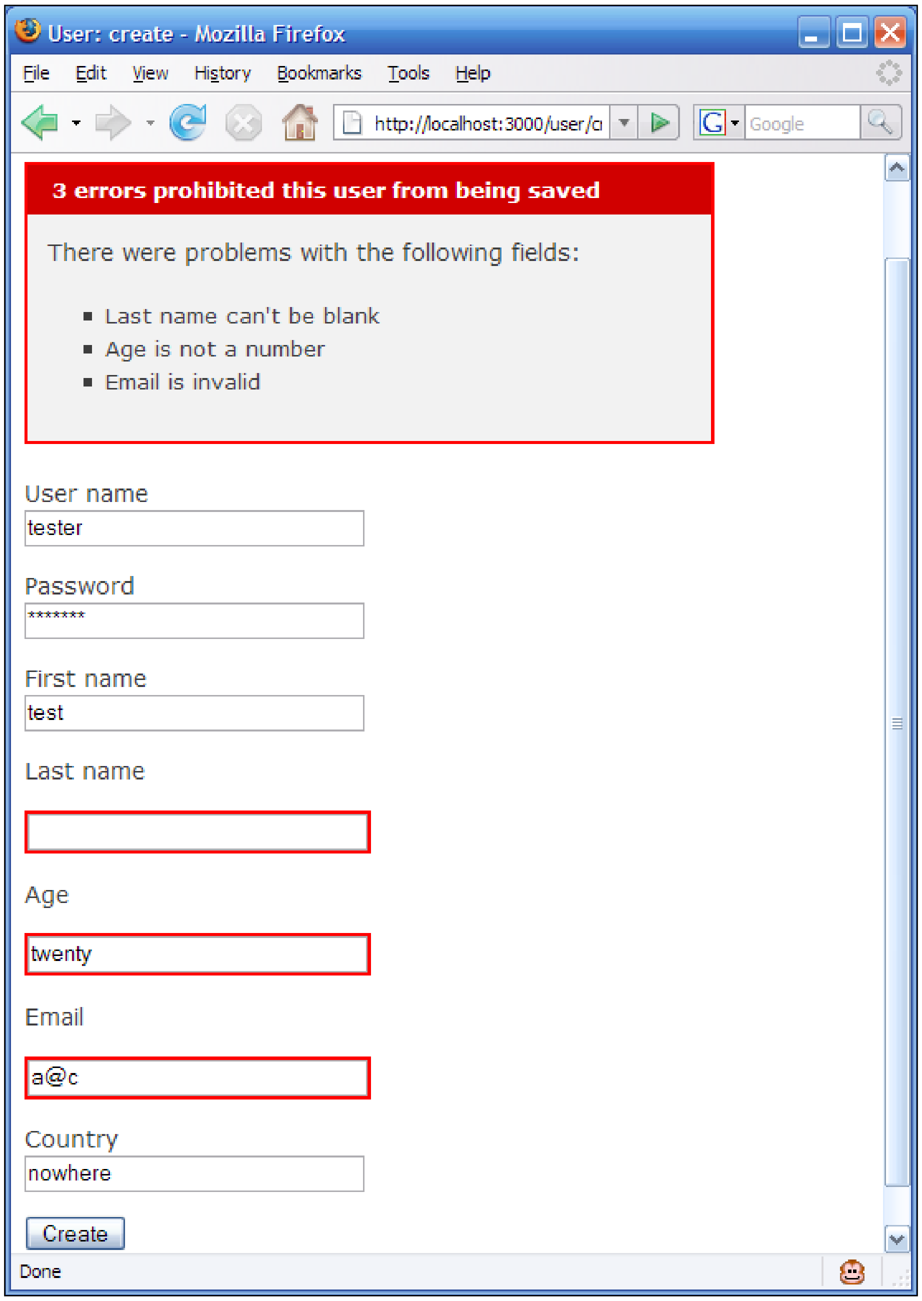
The next functionality we are going to check is the validations on the fields of the 'New User' page. So open up the following URL and go to 'New User' page:

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

Following are the values that we will be giving to check the validations

| **Name** | **Value** |
|---|---|
| User name | **tester** |
| Password | **testing123** |
| First name | **test** |
| Last name | **(leave this blank to test the required field validation)** |
| Age | **twenty** |
| Email | **a@c** |
| Country | **nowhere** |

In the values, **Age** and **Email** validations are what we are really checking for. If you get the following screen, then the validations are working perfectly:


User: create - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/user/c
Google

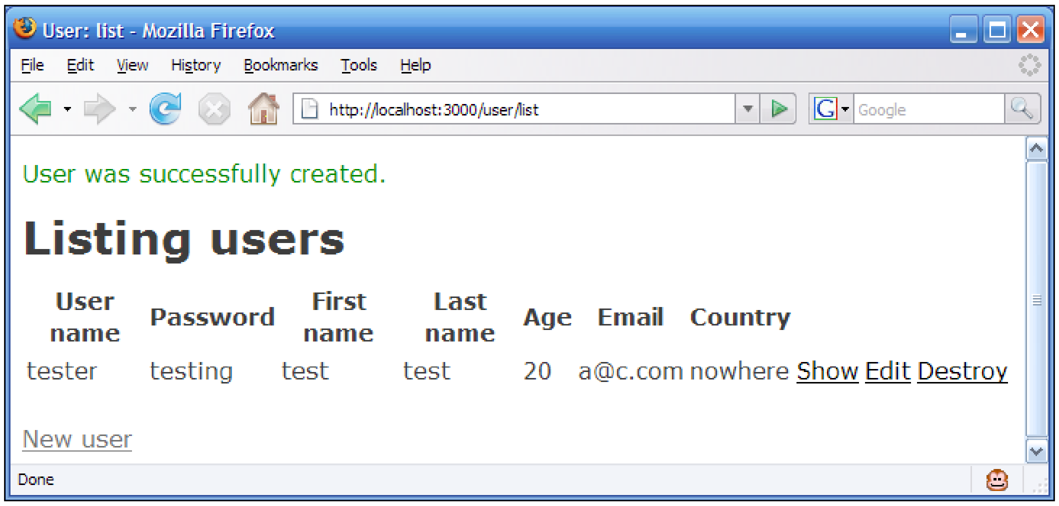
**3 errors prohibited this user from being saved**

There were problems with the following fields:

- Last name can't be blank
- Age is not a number
- Email is invalid

|  |  |
|---|---|
| User name | tester |
| Password | ******* |
| First name | test |
| Last name |  |
| Age | twenty |
| Email | a@c |
| Country | nowhere |

After correcting the values, click on **Create** again. If you get the following screen, then everything is well with the changes we did in the `create` method of `user_controller.rb`.


User: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/user/list
Google
Done

User was successfully created.

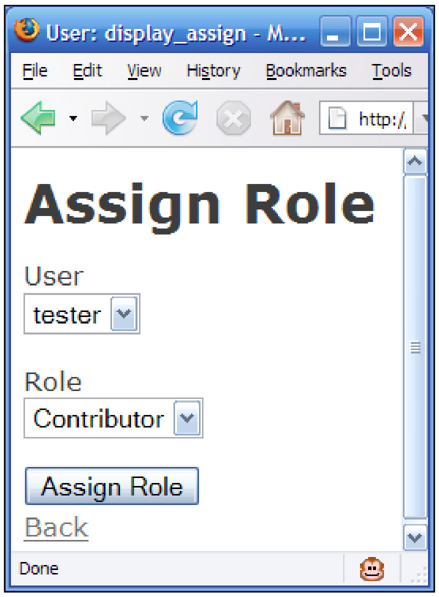
Listing users

| User name | Password | First name | Last name | Age | Email | Country | | | |
|---|---|---|---|---|---|---|---|---|---|
| tester | testing | test | test | 20 | a@c.com | nowhere | Show | Edit | Destroy |

New user

Next, let us test the 'Assign Role' functionality. For that, first add a new **Role** called **Contributor**. Then navigate to the following URL: `http://localhost:3000/user/display_assign`

In response, you will see the following screen:


User: display_assign - M...
File Edit View History Bookmarks Tools
http:/
Done

Assign Role

User

tester

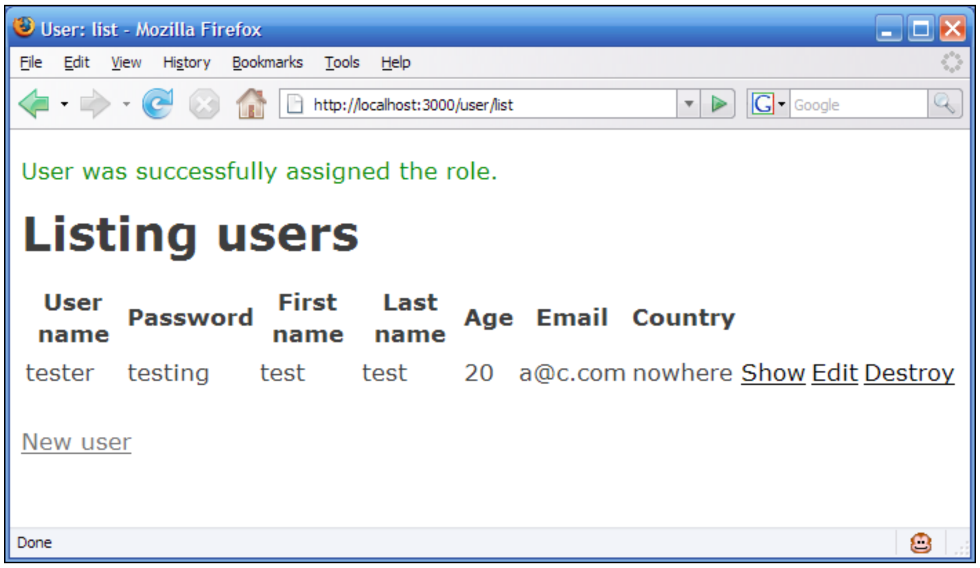
Role

Contributor

Assign Role

Back

Select the **User** whose name is **tester** from the combo box for **User** and **Contributor** from the combo box for the **Role**. Click on **Assign Role** and if you get the following screen then the 'Assign Role' functionality is working perfectly:


User: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/user/list
Google
Done

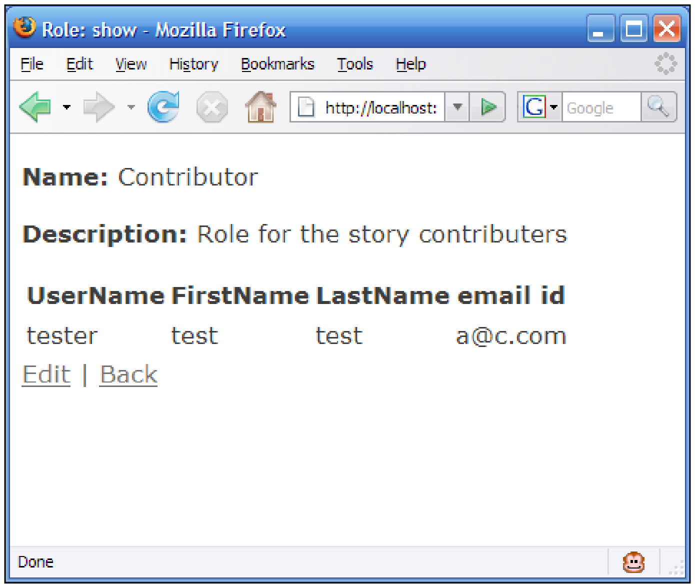
User was successfully assigned the role.

Listing users

| User name | Password | First name | Last name | Age | Email | Country | | | |
|---|---|---|---|---|---|---|---|---|---|
| tester | testing | test | test | 20 | a@c.com | nowhere | Show | Edit | Destroy |

New user

Next, let us test the 'Show details' functionality of Role management. Open the following URL and click on the **Show** link for the **Contributor** Role. You will get the following screen:


Role: show - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:
Google
Done

**Name:** Contributor

**Description:** Role for the story contributers

| UserName | FirstName | LastName | email id |
|---|---|---|---|
| tester | test | test | a@c.com |

Edit | Back

That completes the testing of the functionalities. It tells us that our changes and additions are working fine. As I had mentioned in the beginning of this chapter, the administrator will be looking after the site. Creation of the administrator is left to you as an exercise. The difference between a normal user and an administrator is that an administrator will have the role name Administrator. So, the exercise is to add a Role having Administrator as the name, add a user, and assign Administrator Role to the newly created user.

## Summary

With that, we have set up the User Management for TaleWiki. However, there are some gaps. User management, without User Authentication and Session Management, is like a story half told. Also, due to the lack of Session Management, the changes we did to the tales table could not be implemented at the application level. Apart from these, we are still using the default template provided by RoR. These gaps are what we will address in the next chapter. Keep on reading.

# 5
# Gathering User Comments

In the last chapter, we saw how to set up User Management and Role Management for TaleWiki. However, we did not set up the Login Management based on Users. So, it was work only half done. To complete the task, we will set up Login Management in this chapter. It will not only authenticate a user but also provide the session management.

Secondly, we will look at how to gather user comments for a particular story. We will start with the functionalities to be provided by the Comment Gathering module. We will then move on to the database design for the module. After that we will not only set up the Login Management but also modify the Tale Management so that the User and Tales can be related. We will wrap up with the implementation of the Comment Gathering module. Let's gets started.

## Understanding the Requirements

In this chapter, we will be tackling two problems—managing the user authentication as well as the session management and accepting comments from other users for a particular tale. So we can divide the requirements into two:

- Login Management
- Comment management

The Login Management module will also provide the solution to the problem of Tale management that evolved during the development of User management. As the tales table refers to the users table, without a user id a new tale cannot be submitted. The Login management will provide us the user id corresponding to the new tales. Also, it will tell us who has commented on a particular tale. Let us see how.

## Login Management

As the name suggests, the main functionality the Login Management will provide will be managing the logins. However, managing logins is not a single task. It is dependent on others tasks or operations as well. So, the overall functionalities we will be developing as part of Login management are:

- **Authenticating the User**: We can allow only the registered users to access the functionalities of TaleWiki. This operation will ensure that the user is a registered user before he or she tries to enter the TaleWiki.
- **Setting the Session**: Once the user is found to be authentic, then we have to *maintain his/her authenticity* until he/she logs out. The authenticity can be maintained by this functionality.
- **Checking Roles**: Each User is assigned a Role. So we will need to check whether a particular functionality—such as viewing details of another user—is a part of the Role. This functionality will check the User's Role whenever he/she tries to access any functionality provided by TaleWiki.
- **Invalidating Session**: When a user logs out, all the details of the user in the current session need to be cleared out. This functionality will clear out all the details of the user, including whether the user is authentic or not.

Now that we have defined the functionalities of Login management, let us move on to the next set of tasks—managing the comments.

## Managing the Comments

It is natural for a person to comment upon whatever he or she reads. So, it is necessary to provide a way for users to comment on a particular story. The comments can be of two types—*threaded* and *non-threaded*. In *threaded* comments, one comment can be posted as a response for another comment. If the first comment is removed, then all its child comments will also be removed. If we go for *non-threaded* comments, then each comment is considered an individual. So if one is deleted, others are not affected.

The Comment Management module will do the same. The functionalities that the Comment Management module will provide are:

- **Adding a Comment**: When a user wants to comment on a particular story, he or she can use this functionality. A user can comment on many stories. Comments are not *threaded*. That means a comment cannot be a response for another comment. Each comment is considered an individual.

- **Deleting a Comment:** If an administrator finds a comment offensive or feels that comments are very old, this functionality can be used to delete such comments. Only the administrator will have access to this functionality.
- **Viewing Comments**: Using this functionality, a user can read all the comments submitted for a particular story. It will be available for all users. In addition, the comments will be shown in the *list* view and the *details* view. In *list* view, the comments will be shown for each story, and in the details view, all the details including the date and complete text of the comment will be shown.

We are not providing a way to modify a posted comment. That is because comments are considered one time and brief view of what the user thinks. Hence, no functionality will be provided for the modification of comments. That wraps up the requirements of the Login and Comment Management modules. Next, let us work on the database design for the modules.

## Designing the Database

As you would have already guessed, our next step will be designing the database. However, unlike the modules that we developed previously, we will be designing the database only for one of the two modules. The Login management module doesn't require a table because its functionalities are based on the users and roles tables. So we will have to design the table for the Comment management module only. Just like the previous chapter, the steps for designing the database are:

- Designing  the E-R Model
- Deriving the Schemas
- Creating the Tables

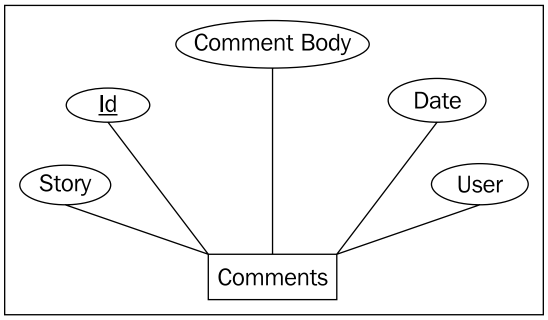
Whenever a new module is added, some of the existing E-R models need to be refined, and consequently the corresponding schemas and tables will be changed accordingly. In the case of Comment management, this holds true as you will see as we go through the steps. So here we go.

## Designing the E-R Model

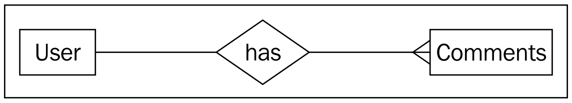
As the name suggests, the Comment Management module will have data related to the comments submitted by the user. What is this data apart from the comment itself? To answer this, first let us try to define the functionality of the Comment Management module in one line. 'Comment management will manage comments submitted by a user for a particular story'—that's how what will look like. The important point here is 'comments submitted by a user for a particular story'. We have three main entities—Comments, Users, and Stories. Story and User entities have already been discussed in Chapters 3 and 4. So let us look at the Comments entity. The attributes for comments will include the date on which the comment has been added and the title of the comment. In short, the Comments entity will have the following attributes:

- Id—the unique number to identify each comment
- Comment body—the text of the comment
- Date—the date on which comment was added
- User—the user who has added the comment
- Story—the story on which the comment has been made

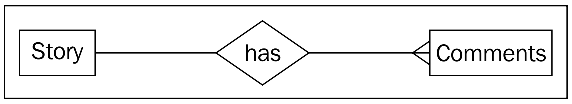
The entity diagram for the Comments entity will be as follows:



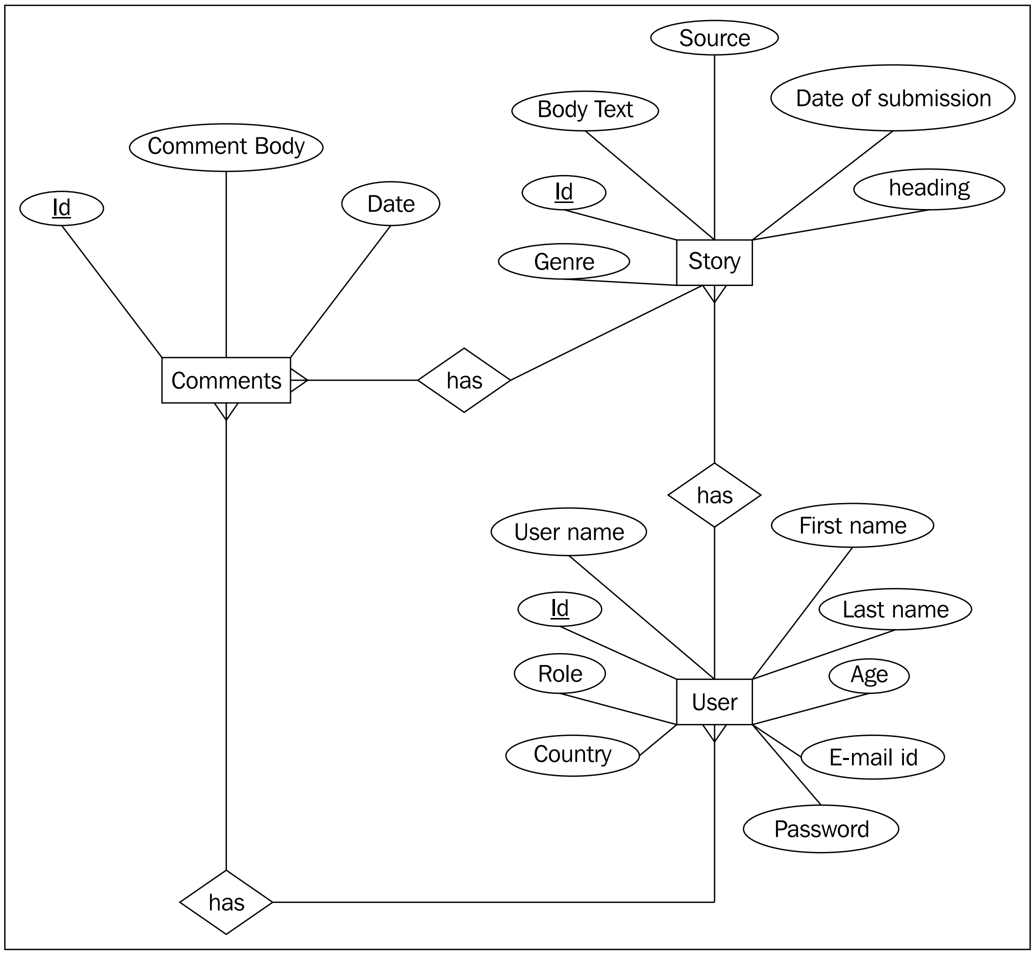
Coming back to our one line definition, we know that the User, Story, and Comments entities are related. The question is how are they related? The answer is there in the one line definition itself. First, let us consider the User entity. The definition says 'comments submitted by a user'. That means one user can submit many comments. Hence, the User entity has a one-to-many relationship with the Comments entity. The relationship will be as follows in terms of an E-R diagram:



The next part of the definition tells us 'comments for a story'. This means that one story can have many comments. In other words, the Comments entity is related to the Story entity through a many-to-one relationship. The Story entity will be at the 'one' end and the Comments entity will be at the 'many' end of the relationship. The diagram will look like as follows:



When looking at all the entities with their attributes and relationships, the picture will be as follows:



The next step obviously is deriving the schema. Here it comes.

## Deriving the Schema

We have the complete information about the attributes and relationships of the Comments entity. The main point about this entity is that unlike the User entity it doesn't introduce any changes in the existing schemas. The reason is that the Comment entity is dependent on other entities and not vice versa. The schema will be as follows:

| Attribute | Data type of the attribute | Length of the acceptable value |
|---|---|---|
| Id | Integer | 10 |
| Comment body | Varchar | 1000 |
| Date of comment | Date | |
| Id of user | Integer | 10 |
| Id of Story | Integer | 10 |

Here Story and User both have their own schemas. So their Ids will be the foreign keys in the table. Now, we can develop the table.

## Creating the Tables

There is only one table to be created. Apart from the attributes, the comments table (keeping with the naming convention), will have two foreign key references—one to the users table and another to the tales table. Including these, the SQL query will be as follows:

```
CREATE TABLE `comments` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`comment_body` TEXT NOT NULL ,
`submission_date` DATE NOT NULL ,
`tale_id` INT NOT NULL,
`user_id` INT NOT NULL,
CONSTRAINT `fk_comments_users` FOREIGN KEY (`user_id`) REFERENCES
users( `id`) ,
CONSTRAINT `fk_comments_tales` FOREIGN KEY (`tale_id`) REFERENCES
tales( `id`)
) ENGINE = innodb;
```

That completes the table definition. By this time you will have started to think that if RoR is so productive, why do we still have to use the SQL statements to create tables?. There is another way—the Ruby way. We will see that in the next chapter where we will convert all the table creation statements using Ruby. Now that the required table has been defined, let us develop the modules starting with the Login management.

# Developing the Login Management Module

Even though Login and session handling are separate functionalities from User management, they depend on the same table—user. Also, the functionalities are more alike than different. Hence, instead of creating a new Controller, we will be using the `UserController` itself as the Controller for the Login module. Keeping this point in mind, let us look at the steps involved in developing the Login Management, which are:

- Creating the Login page
- Implementing the Authentication Method
- Setting up the Session
- Applying Authorization

Leaving aside the first step, all other steps mainly focus on the Controller. Here we go.

## Creating the Login Page

We need a login page with textboxes for user name and password in which users can put their credentials and submit to the login authenticator (fancy name for the action method that will contain the logic to authenticate the user). That's what we are going to create now. The convention for any website is to show the login page when the user enters the URL without any specific page in mind. RoR also follows this convention. For example, if you enter the URL as `http://localhost:3000/user`, it displays the list of users. The reason is that the `index` action method of the `UserController` class calls the `list` method whenever the aforementioned URL is used. From this, we can understand two things—first, the default action method is `index,` and second, the first page to be shown is changeable if we change the `index` method.

What we need is to show the login page whenever a user enters the URL `http://localhost:3000/user`. So let's change the index method. Open the `user_controller.rb` file from the `app/views/user` folder and remove all the statements from the body of the `index method` so that it looks like as follows:

```
def index
end
```

Next, let us create an `index.rhtml` file, which will be shown when the `index method` is called. This file will be the login page. In the `app/views/user` folder, create an `index.rhtml` file. It will be as follows:

```
<%= form_tag :action=> 'authenticate'%>
   <table >
      <tr align="center" class="tablebody">
         <td>User name:</td>
         <td><%= text_field("user", "user_name",:size=>"15" ) %></td>
     </tr>
     <tr align="center" class="tablebody">
        <td>Password:</td>
        <td><%= password_field("user",
                              "password",:size=>"17" ) %></td>
     </tr>
```

```
        <tr align="center" class="tablebody">
           <td></td>
           <td><input type="submit" value=" LOGIN " /></td>
        </tr>
      </table>
```

It uses two new form helpers—`text_field` and `password_field`. The `text_field` creates a text field with the name passed as the parameter, and the `password_field` creates a password field again with the name passed as the parameter. We have passed the `authenticate` method as the action parameter so that the form is submitted to the `authenticate` method. That completes the login page creation. Next, we will work on the `authenticate` method.

## Implementing the Authenticate method

Authenticating a user essentially means checking whether the user name and password given by the user corresponds to the one in database or not. In our case, the user gives us the user name and password through the login page. What we will be doing is checking whether the user is in database and does the password that we got corresponds to the password stored in the database for the user? Here, we will be working on two levels:

- Model
- Controller

We can put the data access part in the action method that being the Controller itself. But it will create problems in the future if we want to add something extra to the user name/password checking code. That's why we are going to put (or delegate) the data access part into Model.

### Model

We will be modifying the `User` class by adding a method that will check whether the user name and password provided by the user is correct or not. The name of the method is `login`. It is as follows:

```
def self.login(name,password)
     find(:first,:conditions => ["user_name = ? and password =
          ?",name, password])
end
```

It is defined as a singleton method of the User class by using the self keyword. The singleton methods are special class-level methods. The conditions parameter of the find method takes an array of condition and the corresponding values. The find method generates an SQL statement from the passed parameters. Here, the find method finds the first record that matches the provided user_name and password. Now, let us create the method that the Controller will call to check the validity of the user. Let us name it check_login. The definition is as follows:

```
def check_login
       User.login(self.user_name, self.password)
end
```

This function calls the login method. Now if you observe closely, check_login calls the login function. One more point to remember—if a method 'test' returns a value and you call 'test' from another method 'test1,' then you don't need to say 'return test' from within 'test1'.The value returned from 'test' will be returned by 'test1' implicitly. That completes the changes to be done at the Model level. Now let us see the changes at the Controller-level.

## Controller

In the Controller for User—UserController—add a new method named authenticate. The method will first create a User object based on the user name and password. Then it will invoke check_login on the newly created User object. If check_login is successful, that is, it does not return nil, then the user is redirected to the list view of Tales. Otherwise, the user is redirected to the login page itself. Here is what the method will look like:

```
def authenticate
         @user = User.new(params[:user])
         valid_user = @user.check_login
         if logged_in_user
            flash[:note]="Welcome "+logged_in_user.name
            redirect_to(:controller=>'tale',:action => "list")
         else
         flash[:notice] = "Invalid User/Password"
         redirect_to :action=> "index"
         end
end
```

The `redirect_to` method accepts two parameters—the name of the Controller and the method within the Controller. If the user is valid, then the `list` method of `TaleController` is called, or in other words, the user is redirected to the list of tales. Next, let us make it more robust by checking for the `get` method. If a user directly types a URL to an action, then the `get` method is received by the method. If any user does that, we want him/her to be redirected to the login page. To do this, we wrap up the user validation logic in an `if/else` block. The code will be the following:

```
def authenticate
      if request.get?
         render :action=> 'index'
      else
      @user = User.new(params[:user])
         valid_user = @user.check_login
         if valid_user
            flash[:note]="Welcome "+valid_user.user_name
            redirect_to(:controller=>'tale',:action => 'list')
         else
            flash[:notice] = "Invalid User/Password"
            redirect_to :action=> 'index'
         end
      end
end
```

The `get?` method returns true if the URL has the `GET` method else it returns false. That completes the login authentication part. Next, let us set up the session.

In Ruby, any method that returns a Boolean value—true or false—is suffixed with a question mark (`?`). The `get` method of the request object returns a boolean value. So it is suffixed with a question mark (`?`).

## Setting up the Session

Once a user is authenticated, the next step is to set up the session to track the user. Session, by definition, is the conversation between the user and the server from the moment the user logs in to the moment the user logs out. A conversation is a pair of requests by the user and the response from the server. In RoR, the session can be tracked either by using cookies or the session object. The session is an object provided by RoR. The session object can hold objects where as cookies cannot. Therefore, we will be using the session object. The session object is a hash like structure, which can hold the key and the corresponding value. Setting up a session is as easy as providing a key to the session object and assigning it a value. The following code illustrates this aspect:

```
def authenticate
       if request.get?
             render :action=> 'index'
       else
          @user = User.new(params[:user])
          valid_user = @user.check_login
          if valid_user
                session[:user_id]=valid_user.id
                flash[:note]="Welcome "+valid_user.user_name
                redirect_to(:controller=>'tale',:action => 'list')
          else
                flash[:notice] = "Invalid User/Password"
                redirect_to :action=> 'index'
          end
       end
end
```

That completes setting up the session part. That brings us to the last step—applying authorization.

## Applying Authorization

Until now, we have authenticated the user and set up a session for him/her. However, we still haven't ensured that only the authenticated users can access the different functionalities of TaleWiki. This is where authorization comes into the picture. Authorization has two levels—*coarse grained* and *fine grained*. *Coarse grained* authorization looks at the whole picture whereas the *fine grained* authorization looks at the individual 'pixels' of the picture. Ensuring that only the authenticated users can get into TaleWiki is a part of *coarse grained* authorization while checking the privileges for each functionality comes under the *fine grained* authorization. In this chapter, we will be working with the coarse grained authorization.

The best place to apply the *coarse grained* authorization is the Controller as it is the central point of data exchange. Just like other aspects, RoR provides a functionality to easily apply any kind of logic on the Controller as a whole in the form of filters. To jog your memory, a filter contains a set of statements that need to be executed before, after (or before and after) the methods within the Controllers are executed.

Our problem is to check whether the user is authenticated or not, before any method in a Controller is executed. The solution to our problem is using a 'before filter'. But we have to apply authorization to all the Controllers. Hence, the filter should be callable from any of the Controller. If you look at the definition of a Controller, you can find such a place. Each Controller is inherited from the `ApplicationController`. Anything placed in `ApplicationController` will be callable from other Controllers. In other words, any method placed in ApplicationController becomes global to all the Controllers within your application. So, we will place the method containing the filter logic in `ApplicationController`.

To check whether a user is authentic or not, the simplest way is to check whether a session exists for that person or not. If it exists, then we can continue with the normal execution. Let us name it `check_authentic_user`. The implementation will be as follows:

```
def check_authentic_user
         unless session[:user_id]
         flash[:notice] = "Please log in"
         redirect_to(:controller => "user", :action =>
                                    "index")
      end
end
```

It checks for the `user_id` key in a session. If it is not present, the user is redirected to the login page. Place the code in the `application.rb` file as a method of `ApplicationController`. Next, let us use it as a filter. First, we will tell `UserController` to apply the filter for all the action methods except `index` and `authenticate` methods. Add the following statement to the `UserController`. It should be the first statement after the starting of the `Controller` class.

```
class UserController < ApplicationController
before_filter :check_authentic_user, :except =>[ :index, :authenticate
]

:
:
end
```

Similarly, we will place the filter in other Controllers as well. However, in their case, there are no exceptions. So `TaleController` will have:

```
class TaleController < ApplicationController
before_filter :check_authentic_user

:
:
end
```

`GenreController` and `RoleController` will be the same as `TaleController`. Thus, we have completed the 'applying authorization' part for the time being. Now, let's tie up one loose end—the problem of adding a new tale.

## Tying Up the Loose Ends

When we developed the User management, the Tale management was affected as the tales table has a many-to-one relationship with the users table. Now we can solve the problem created by the foreign key reference. First, open the `user.rb` file and add the following statement indicating that it is at the 'one' end of the relationship:

```
has_many :tale
```

After addition of the statement, the class will look like the following:

```
class User < ActiveRecord::Base
  validates_presence_of :user_name, :password, :first_name,
  :last_name, :age, :email, :country
  validates_uniqueness_of :user_name
  validates_numericality_of :age
  validates_format_of :email, :with => /\A([^@\s]+)@((?:[-a-
  z0-9]+\.)+[a-z]{2,})\Z/i
  belongs_to :role
  has_many :tale

  def check_login
    User.login(self.name, self.password)
  end
def self.login(name,password)
    find(:first,:conditions => ["user_name = ? and password
        =?",name, password])
end
end
```

Next, add the following statement to the `tale.rb` file:

```
belongs_to :user
```

The file will look like as follows:

```
class Tale < ActiveRecord::Base
  validates_presence_of :title, :body_text, :source
  belongs_to:genre
  belongs_to :user
end
```

Next, open the `tale_controller.rb` file. In the `create` method, we need to add the user's id to the tale's user id reference so that the referential integrity can be satisfied. For that, we will get the current user's id from the session and set it as the value of the user_id attribute of the tale object. The `create` method will look like as follows, after doing the changes:

```
def create
    @tale = Tale.new(params[:tale])
    @tale.genre_id=params[:genre]
    @tale.user_id=session[:user_id]

    @tale.status="new"
    if @tale.save
      flash[:notice] = 'Tale was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
end
```

That's it. The 'loose ends' related to the User management are tied up. Now let us move onto the Comment Management module.

## Developing the Comment Management Module

From the description of functionalities, we know that the module needs to support only three operations—add, view, and delete. The steps for developing the module are almost the same:

- Generating the Scaffold
- Modifying the Model
- Refining the View
- Customizing the Controller

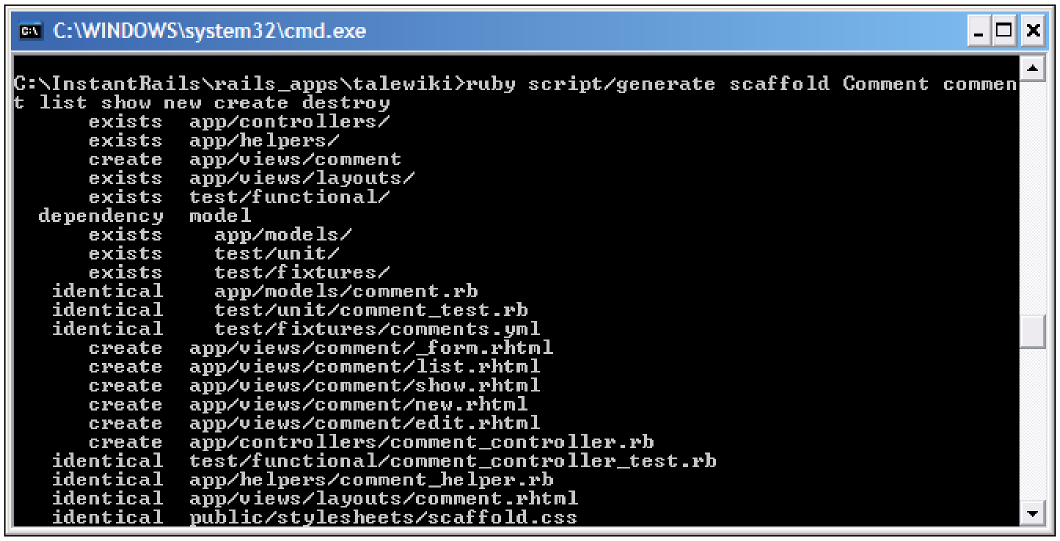
We have changed the order of refining the view and customizing the Controller steps. That's what I meant by 'almost the same'. Let's get into the development.

## Generating the Scaffold

Open the RoR prompt using `use_ruby` command, and enter the following command:

```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold
Comment comment list show new create destroy
```

You will get the following screen:



```
C:\InstantRails\rails_apps\talewiki>ruby script/generate scaffold Comment commen
t list show new create destroy
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/comment
      exists  app/views/layouts/
      exists  test/functional/
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
   identical    app/models/comment.rb
   identical    test/unit/comment_test.rb
   identical    test/fixtures/comments.yml
      create  app/views/comment/_form.rhtml
      create  app/views/comment/list.rhtml
      create  app/views/comment/show.rhtml
      create  app/views/comment/new.rhtml
      create  app/views/comment/edit.rhtml
      create  app/controllers/comment_controller.rb
   identical  test/functional/comment_controller_test.rb
   identical  app/helpers/comment_helper.rb
   identical  app/views/layouts/comment.rhtml
   identical  public/stylesheets/scaffold.css
```

If the scaffold command is reused, then it will not rewrite the existing files unless you specify the `-force` parameter. We need only `new`, `list`, and `delete` functionalities. So, we have specified the actions that we need—`list`, `show` for listing of comments, `new` and `create` for adding, and `delete` for deleting. However, it will still create the stubs and links that need to be tackled at the View level. First, let us do the required modifications at the Model level.

## Modifying the Model

First, we have to tell RoR which fields should not be empty. For that, add the `validates_presence_of` method with `:comment_body` as the argument in the `comment.rb` file. After addition, the code shall be as follows:

```
class Comment < ActiveRecord::Base
  validates_presence_of :comment_body
end
```

Next, we have to tell that the comments table is at the 'many' end of the relationship with both tales and users table. For that, add a `belongs_to` declaration to the `comment.rb` file.

```
class Comment < ActiveRecord::Base
  validates_presence_of :comment_body
  belongs_to :tale
  belongs_to :user
end
```

The next step is to tell both the users and the tales table that they are at the 'one' end of the relationship. For that, open the `user.rb` and `tale.rb` files, and add the `has_many` declaration. After the additions, the code will be as follows for `user.rb`:

```
class User < ActiveRecord::Base
  validates_presence_of :user_name, :password, :first_name,
  :last_name, :age, :email, :country
  validates_uniqueness_of :user_name
  validates_numericality_of :age
  validates_format_of :email, :with => /\A([^@\s]+)@((?:[-a-
  z0-9]+\.)+[a-z]{2,})\Z/i
  belongs_to :role
  has_many :tale
  has_many :comment
  def check_login
     User.login(self.name, self.password)
  end
  def self.login(name,password)
     find(:first,:conditions => ["user_name = ? and password
        =?",name, password])
  end
end
```

For `tale.rb`, here is the code:

```
class Tale < ActiveRecord::Base
  validates_presence_of :title, :body_text, :source
  belongs_to:genre
  belongs_to :user
  has_many :comment
end
```

That completes the changes to be done at the Model level. Next, let us refine the View.

## Refining the View

Comments will be given for a story. That means the page displaying a tale will have a link to add comments. This also means that the Comment management module is not a 'standalone' module like others, as it will not have its own menu when we decide upon the template. Now coming back to links to the comments in the tale display page, for what functionalities do we need the links? The answer is two—adding a comment and listing the comment. The add comment link will lead to the 'New Comment' page, and the view comments link will lead to the list view of the comments. Now let us see what are the problems—each comment needs a user id and the id of the tale for which the comment is being added. The listing of comments needs only the id of the tale. As user id is available from the session, we have to add only the tale id as a part of the link. That is what we are going to do.

Open the `show.rhtml` file from the `app/views/tale` directory. It contents are as follows

```
<% for column in Tale.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @tale.send(column.name) %>
</p>
<% end %>
<%= link_to 'Edit', :action => 'edit', :id => @tale %>
<%= link_to 'Back', :action => 'list' %>
```

Now let us add two more links—one for adding a comment and another for listing the comments:

```
<% for column in Tale.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @tale.send(column.name) %>
</p>
<% end %>

<%= link_to 'Edit', :action => 'edit', :id => @tale %>
<%= link_to 'Back', :action => 'list' %>
<%= link_to 'Add Comment',:controller=>'comment', :action => 'new', :
id => @tale.id %>
<%= link_to 'View Comments',:controller=>'comment', :action => 'list',
:id => @tale.id %>
```

The next change we have to do is remove the `edit` option from the viewing part of the comments. So open the `list.rhtml` file from the `app/views/comments`. The code will be as follows:

```
<h1>Listing comments</h1>

<table>
  <tr>
  <% for column in Comment.content_columns %>
    <th><%= column.human_name %></th>
  <% end %>
  </tr>

<% for comment in @comments %>
  <tr>
  <% for column in Comment.content_columns %>
    <td><%=h comment.send(column.name) %></td>
  <% end %>
    <td><%= link_to 'Show', :action => 'show', :id => comment %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => comment %></td>
    <td><%= link_to 'Destroy', { :action => 'destroy', :id => comment
}, :confirm => 'Are you sure?', :method => :post %></td>
  </tr>
<% end %>
</table>

<%= link_to 'Previous page', { :page => @comment_pages.current.
previous } if @comment_pages.current.previous %>
<%= link_to 'Next page', { :page => @comment_pages.current.next } if @
comment_pages.current.next %>

<br />

<%= link_to 'New comment', :action => 'new' %>
```

Delete the tags that link to the `Edit` and `New Comment` functionalities. We do not need anyone adding a comment without reading the story. After deletions, the code will be as follows:

```
<h1>Listing comments</h1>

<table>
  <tr>
  <% for column in Comment.content_columns %>
    <th><%= column.human_name %></th>
  <% end %>
  </tr>

<% for comment in @comments %>
  <tr>
  <% for column in Comment.content_columns %>
```

```
    <td><%=h comment.send(column.name) %></td>
  <% end %>
    <td><%= link_to 'Show', :action => 'show', :id => comment %></td>
    <td><%= link_to 'Destroy', { :action => 'destroy', :id => comment
}, :confirm => 'Are you sure?', :method => :post %></td>
  </tr>
<% end %>
</table>
<%= link_to 'Previous page', { :page => @comment_pages.current.
previous } if @comment_pages.current.previous %>
<%= link_to 'Next page', { :page => @comment_pages.current.next } if @
comment_pages.current.next %>
<br />
```

That completes the refinement to be done to the VIEW. Now let's modify the Controller.

## Customizing the Controller

Open the `comment_controller.rb` file and in the `new` method add the `tale_id` to the `session` object so that the method looks like the following:

```
def new
   @comment = Comment.new
   session[:tale_id]=params[:id]
end
```

Now in the `create` method, let us get the `tale_id` and the `user_id` from the `session`, and pass it to the `comment` object. We have used the `session` object because the tale_id is coming as a part of the get request, which will be available only to the `new` method and not the `create` method. After the changes, the `create` method will be as follows:

```
def create
    @comment = Comment.new(params[:comment])
    @comment.tale_id=session[:tale_id]
    @comment.user_id=session[:user_id]
    if @comment.save
      flash[:notice] = 'Comment was successfully created.'
      redirect_to  :action => 'list'
    else
      render :action => 'new'
    end
  end
```

We do not want to show the list of comments, once a comment has been added. Therefore, we will redirect the user to the tale's list once a comment has been added successfully.

```
def create
    @comment = Comment.new(params[:comment])
    @comment.user_id=session[:user_id]
    @comment.tale_id=session[:tale_id]
    if @comment.save
      flash[:notice] = 'Comment was successfully created.'
      redirect_to :controller=>'tale', :action => 'list'
    else
      render :action => 'new'
    end
  end
```

Apart from this, we have to change the `list` method so that it finds that only those comments are selected for which the `tale_id` has been passed through the link. So let us modify the `paginate` method in the `list` method to add a condition. After modification, the `list` method will be as follows:

```
def list
    @comment_pages, @comments = paginate :comments, :
conditions=>['tale_id = ?',
 params[:id]] :per_page => 10
end
```

As you can see, the `paginate` method takes the table to paginate, the condition which is optional and the number of items to be shown per page as arguments.

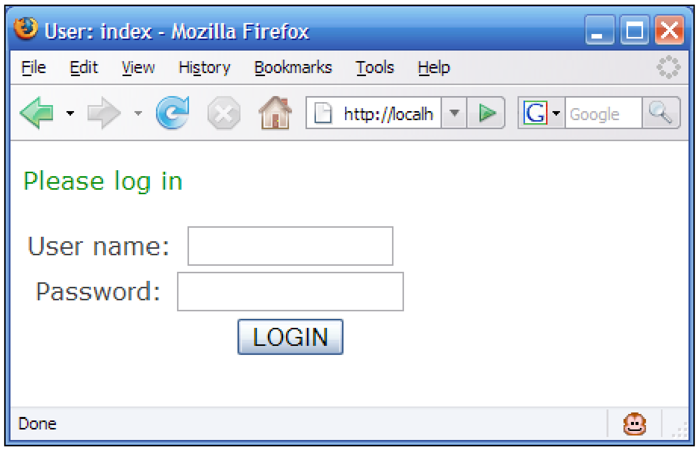
And that completes our current work on the Comment management module. Now it is testing time!

## Testing the Module

Let us start with the authorization part. Give the following URL at the address bar:

`http://localhost:3000/tale`

If you get the following screen, it means authorization is working fine:

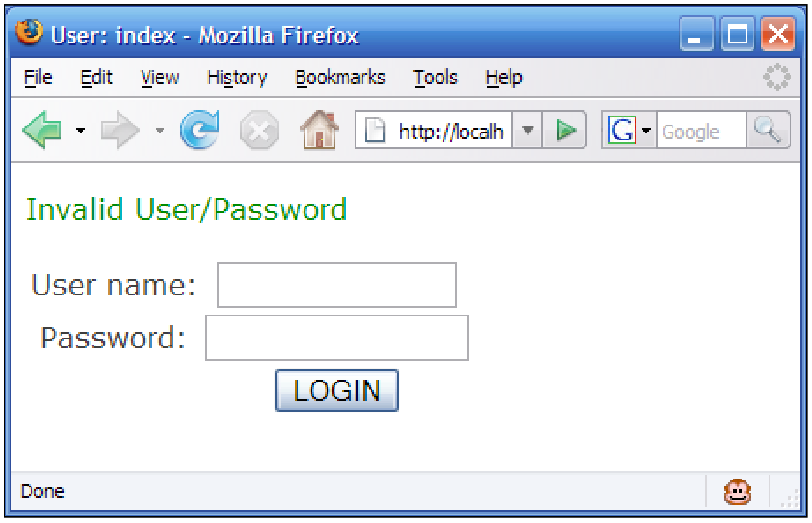

User: index - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localh
Google
LOGIN
Done

Please log in

User name:

Password:

Next let us test the login functionality. Firstly, give the wrong **User name** and **Password** (give anything). If you get the following screen, then the changes are working fine:

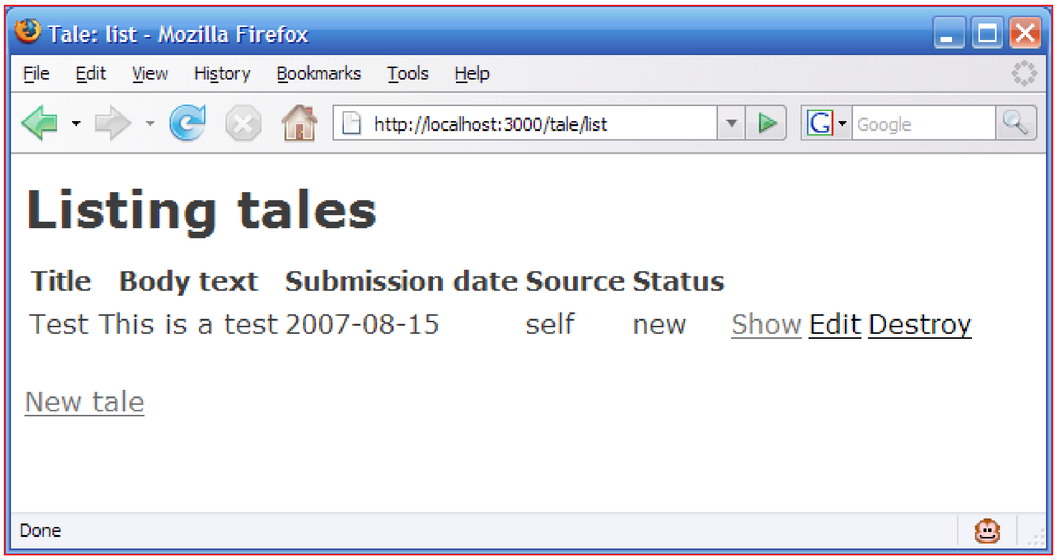

User: index - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localh
Google
LOGIN
Done

Invalid User/Password

User name:

Password:

Now, give the correct **User name/Password** combination. I am giving **tester** as **User name** and **testing** as password. If you get the following screen, then authentication is working fine, and also the redirection is doing what it is supposed to do.

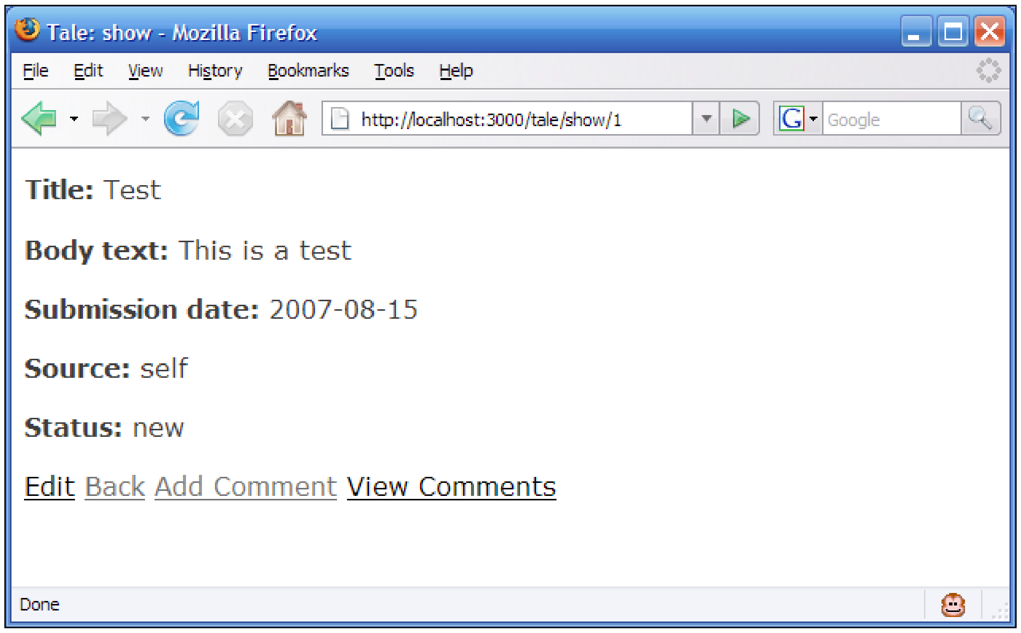

Tale: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/tale/list
Google
Done

# Listing tales

| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show Edit Destroy |

New tale

Now click on the **list** link of the first tale and you will get the following screen:


Tale: show - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/tale/show/1
Google
Done

**Title:** Test

**Body text:** This is a test

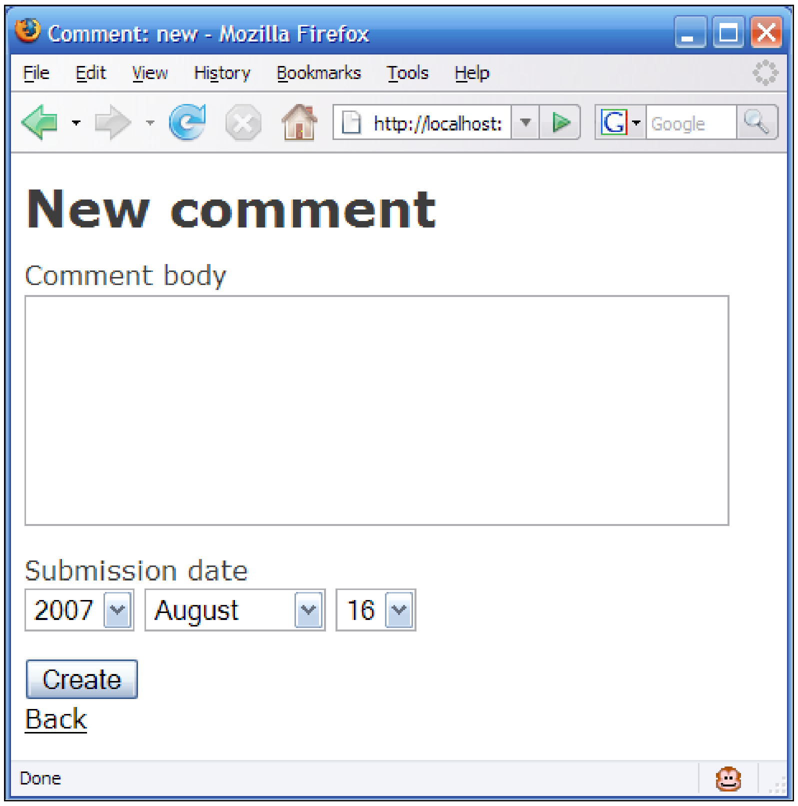
**Submission date:** 2007-08-15

**Source:** self

**Status:** new

Edit Back Add Comment View Comments

On the detail page, click on the **Add Comment** link. The following screen will be displayed:


Comment: new - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:
Google
Create
Back
Done

New comment

Comment body

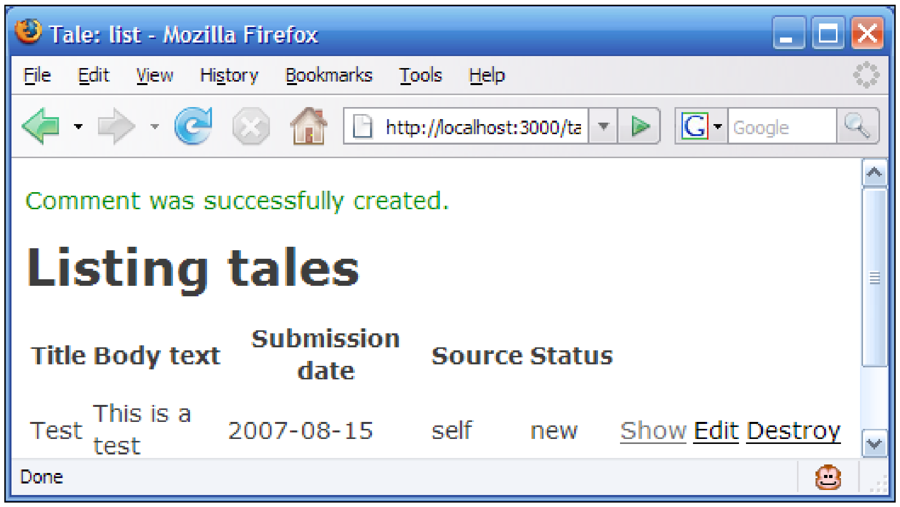
Submission date
2007 August 16

Give the following inputs:

**Comment Body**—**This is a test**.

**Submission Date**—**(leave the default date)**

Now click on **Create**. Then, if you get the following screen you can rest assured that everything is working as planned.

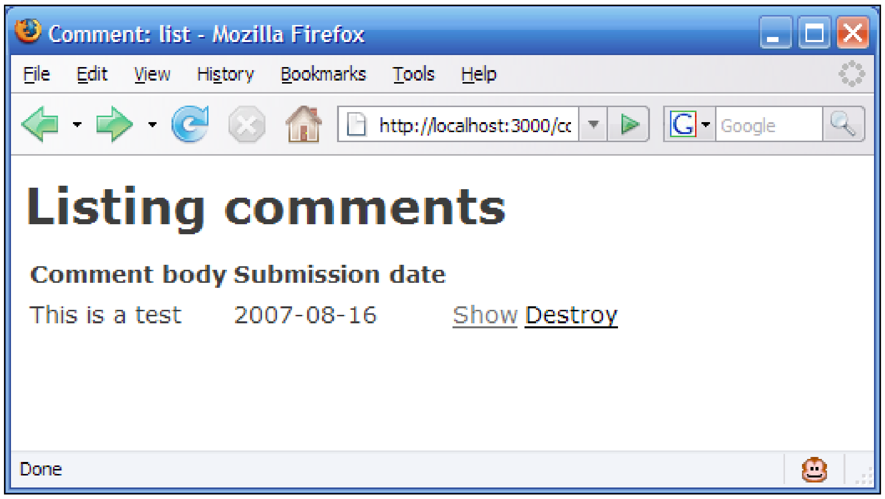

Tale: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/ta
Google
Done

Comment was successfully created.

Listing tales

| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show Edit Destroy |

Now click on the **Show** link again and select the **View Comments** link. If you get the following screen, then the functionality is working:


Comment: list - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/cc
Google
Done

Listing comments

| Comment body | Submission date | |
|---|---|---|
| This is a test | 2007-08-16 | Show Destroy |

These tests tell us that the changes we did are working fine. And that completes our 'endeavour' on gathering the user comments.

## Summary

We have completed Login management and Comment management. Login management was one of the loose ends from the User management part. Now we can concentrate on enhancing the developed modules. These enhancements include custom template creation, the logout option, database-independent table creation, and other features that need to be completed before moving on to developing the new functionalities. These enhancements will implemented in the next chapter. So keep reading!

# 6 Setting up the Template

In the previous chapters, you have seen the steps involved in creating an initial setup for each module and customizing the setup to meet our needs. However, we are still using the default template that RoR provided with the initial setup. The default template is the only template provided by RoR. Moreover, currently each module is standalone, that is, the user has to give the complete URL to the module to access it. To overcome this situation, we need to set up a navigation system that will link the modules with each other at the 'View level'. Thus, the user will be able to access the different functionalities that we have developed without providing the complete address to the page providing the functionality.

The other matter of concern is the creation of the database. Until now, we have created the table using SQL. The problem with this approach is that it has locked us down with MySQL. If we want to shift to the Oracle server, all the table creation scripts will have to be rewritten. RoR provides a solution to this problem in the form of Migration.

In this chapter, our focus will be on two things—setting up the template for TaleWiki and generating the migration for our tables. It is always a good idea to start from the back-end (database) and move towards the front-end (template). Therefore, we will create migration for our files and then move onto setting up the template.

## Understanding Migration

The functionality provided by Migration in RoR can be defined as 'Managing the evolution of a schema used by several physical databases using Ruby, making it possible to use a version control system to keep things synchronized with the actual code.' The keywords here are *schema using Ruby* and *keep synchronized with actual code*. In other words, Migration helps you to create and manage tables using Ruby without going into the native SQL. In other words 'The Migration tool records and plays back incremental changes to the database schemas. Data definitions are database-independent. This allows the database to be recreated as it was at any point in the project's lifecycle on any database platform.' Following are some of the tasks that you can accomplish using Migration:

- **Creation of Table**: The `create_table` method allows you to create a table in the database provided in the `database.yml` file. It takes the table name and options as arguments. The options include the name of the column, type of the column, and so on. The options supplied are usually in the form of an anonymous code block providing details of the columns of the table. The options can be varied. That's why the anonymous code block is used.
- **Dropping a Table**: Using `drop_table`, you can drop a table. It takes the name of the table to be dropped as the only argument. Cascade delete will be caused only if the cascade option is provided as a native SQL query.
- **Adding a new Column**: When you need to add a new column to an existing table, you can use the `add_column` method. It takes the table name, the name of the column to be added, data type of the column, and options such as the size of the column as the arguments.
- **Defining an Index**: There are situations where indexing a table on a particular column speeds up the data access. For such situations, `add_index` can be of great help. It accepts the table name, the column name upon which the index needs to be generated, and the type of index as arguments.

Next, let us implement the Migration for TaleWiki. We can divide the implementation process into the following steps:

- Generating Migration classes
- Editing the Generated classes
- Running Migration

I will be implementing migration for two tables—genres and tales. I am leaving the Migration for other tables as an exercise for you.

## Generating Migration Classes

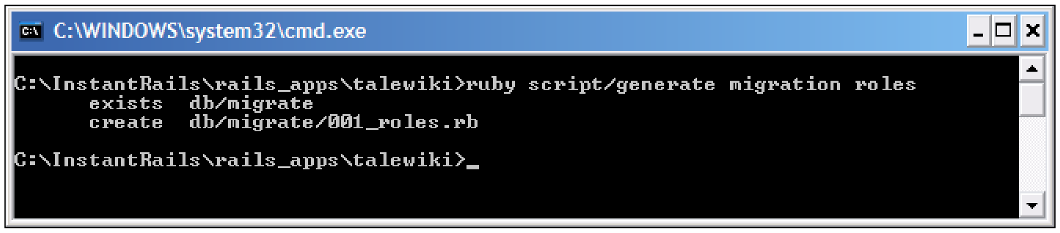
In keeping with the philosophies of RoR, migrations are also generated using the generator supplied with RoR. The generator creates the migration classes (which are Ruby files in the `db/migrate` folder of the rails application). For example, when we generate migration files for our site, they will be created in the `db` folder directly under the `talewiki` folder. The syntax of the migration generator is:

```
ruby script/generate migration  <table_name>
```

Here, `<table_name>` is the name of the table for which the migration file will be generated. To make it more clear, let us create migration for genres and tales tables. Bring up the command prompt and give the `use_ruby` command to set the required environment variables. At the prompt, give the following commands:

```
C:\InstantRails\rails_apps>cd talewiki
C:\InstantRails\rails_app\talewiki>ruby script/generate migration roles
```

You will get the following screen:



```
C:\InstantRails\rails_apps\talewiki>ruby script/generate migration roles
      exists  db/migrate
      create  db/migrate/001_roles.rb

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

Similarly, for the tales table give the following command:

```
C:\InstantRails\rails_app\talewiki>ruby script/generate migration users
```

You will get a screen similar to the one you got for the genres table. If you look into the `db/migrate` folder, you will find two files—`001_roles.rb` and `002_users.rb`. These are the migration files for our tables. Whenever RoR generates the Migration file related to the Migration class, it will append the numerical prefix starting from `001`. The first file generated will have the value `001` prefixed. Let us move on to the next step.

## Editing the Generated Classes

The second step is to edit the generated classes. Open the `001_roles.rb` file. You will find the following code:

```
class Roles < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

The class `Roles` is derived from the `Migration` class. The generated code contains two methods:

- **up**: RoR calls this method to create the table on the database specified in the `database.yml` file. The table creation code will go into this method.
- **down**: RoR calls this method when the changes done to the database need to be rolled back. The code to handle rollback will be part of this method.

Before we move towards the table creation code, let us look at how the SQL data types map to the Ruby data type, which is displayed in the following table:

| SQL Data type | Ruby Data type |
|---|---|
| Varchar | String |
| Int | Integer |
| Decimal | Float |
| Text | Text |
| Date | Datetime |
| BLOB | Binary |

Now let us modify the `self.up` method. As we have to create a table, we will call the `create_table` method here. After modification, the `self.up` and `self.down` methods will be as follows:

```
class Roles < ActiveRecord::Migration
  def self.up
  create_table :roles do |t|
    t.column :name, :string, :limit => 25, :null =>
                   false
    t.column :description, :string, :limit => 100, :null =>
                   false
  end
  def self.down
    drop_table :roles
  end
end
```

The `create_table`, as you will remember, takes the table name and options in the form of an anonymous code block, as arguments. In the block, we are setting the data type, size, and null ability (whether a column can be null or not) for each column of the table. Size is specified with `:limit`. If there is a problem during table creation, we want the table to be dropped. Therefore, in the `self.down`, we are calling the `drop_table` method. Now, let us apply the same step on `002_users.rb`. Open `002_users.rb` and modify the `self.up` and `self.down` so that they look like the following code:

```
class Users < ActiveRecord::Migration
  def self.up
     create_table :users do |t|
        t.column :user_name, :string, :limit => 50, :null =>
                                                     false
        t.column :password, :string, :limit => 15, :null =>
                                                     false
        t.column :first_name, :string, :limit => 50, :null =>
                                                     false
        t.column :last_name, :string, :limit => 50, :null =>
                                                     false
        t.column :age, :int, :limit=>3, :null=>false
        t.column :email, :string, :limit => 25, :null => false
        t.column :country, :string, :limit => 20, :null =>
                                                 false
        t.column :role_id, :int, :limit => 11, :null => false
     end
  end
  def self.down
    drop_table :users
  end
end
```

Next, we have to tell RoR that the users table is at the 'many' end of the 'one-to-many' relationship it has with the roles table. Unfortunately, there is no 'Ruby' way to do that. To create the foreign key reference, you will have to use native SQL by calling the `execute` method and passing the SQL as an argument. This locks us down to a particular vendor. The way around it is to implement the conditional logic after `create_table` to check the vendor of the database. However, the implementation is out of the scope of the current discussion. On adding the foreign key reference, the code will be as follows:

```
class Users < ActiveRecord::Migration
  def self.up
     create_table :users do |t|
        t.column :user_name, :string, :limit => 50, :null =>
                                                    false
        t.column :password, :string, :limit => 15, :null =>
                                                    false
        t.column :first_name, :string, :limit => 50, :null =>
                                                     false
        t.column :last_name, :string, :limit => 50, :null =>
                                                     false
        t.column :age, :int, :limit=>3, :null=>false
        t.column :email, :string, :limit => 25, :null => false
        t.column :country, :string, :limit => 20, :null =>
                                                 false
        t.column :role_id, :int, :limit => 11, :null => false
```

```
            end
            execute "ALTER TABLE user ADD FOREIGN KEY ( role_id )
            REFERENCES roles (id) ON DELETE CASCADE "
         end
         def self.down
           drop_table :users
         end
       end
```

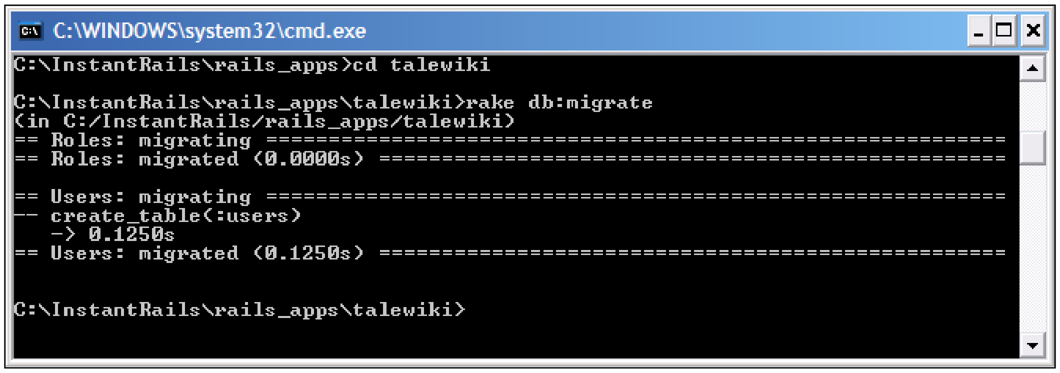
That completes the table creation using Migration. However, if you observe closely, we haven't added the column for primary key in both the tables. Yet, we are referencing the primary key of the roles table in the SQL statement passed to the `execute` method. The reason is that the Migration creates the primary keys automatically in keeping with the conventions of RoR. Now we can move onto step three.

## Running the Migration

The third step is to run or execute the Migration so that the tables are created in the database. To run Migration, we will use a tool called rake. *Rake* is a program for Ruby that is similar to the *make* program. Both are build programs. A build program is a program that simplifies the execution of complex tasks, such as compilation of programs having multiple dependencies, generating the database schema, and so on C/C++ uses *make* whereas RoR *uses* rake. Let us put *rake* to the task for running the migration. At the command prompt, give the following command:

```
C:\InstantRails\rails_app\talewiki>rake db:migrate
```

The first argument tells about the directory in which the migration files are kept, and the second argument—migrate—specifies the task to be executed. The result of the *rake* command will be similar to the following screen:



```
C:\InstantRails\rails_apps>cd talewiki

C:\InstantRails\rails_apps\talewiki>rake db:migrate
(in C:/InstantRails/rails_apps/talewiki)
== Roles: migrating ===========================================================
== Roles: migrated (0.0000s) ==================================================

== Users: migrating ===========================================================
-- create_table(:users)
   -> 0.1250s
== Users: migrated (0.1250s) ==================================================


C:\InstantRails\rails_apps\talewiki>
```

That completes the steps for using Migration. The other tables—tales, genres, and comments are left to you. Next, let us customize the template of our site.

> You can find more about the options and methods that Migration supports from the RoR docs, available at the following URL:
> `http://api.rubyonrails.org/classes/ActiveRecord/Migration.html`

## Customizing the Template

A template can be defined as 'A master page that dictates the look and feel of the whole website.' In other words, a template contains information about the layout as well as placeholders for dynamic data, such as navigation, and menu. The definition provides us with the points on which we will be working, layout and navigation. Therefore, to customize the template of Talewiki, we will be performing the following tasks:

- Defining the Layout
- Setting up the Navigation

Let us look into the details of each task.

## Defining the Layout

For TaleWiki, we require two separate layouts—one for the login page and the other for rest of the TaleWiki pages. The reason is that the login page is, in a way, outside the system and it does not have the menu system. Therefore, we can divide the 'defining the layout' task into:

- Defining Layout for Login Page
- Defining Master Layout

The layout for the login page will be embedded within the login page itself whereas the Master Layout will be a separate RHTML file that will be included wherever we require the specific layout. So, let's get started.

### Customizing the Layout of the Login Page

The layout that we are going to define for the login page will be used by the login page only. Therefore, it will not be kept in a different RHTML page. The current layout does not have a header. So the first thing we will be adding is a header. The header is an image file. Open the `index.rhtml` file from the `app/views/users` folder. The file contains the following code:

```
<%= form_tag :action=>'authenticate' %>
   <table >
       <tr align="center" class="tablebody">
          <td>User name:</td>
          <td><%= text_field("user", "user_name",:size=>"15" ) %></td>
       </tr>
       <tr align="center" class="tablebody">
          <td>Password:</td>
          <td><%= password_field("user", "password",:size=>"17"
           ) %></td>
       </tr>
       <tr align="center" class="tablebody">
          <td></td>
          <td><input type="submit" value=" LOGIN " /></td>
       </tr>
   </table>
```

First, enclose the page in the `<body>` and `<html>` tags so that the source will be as follows:

```
<html>
<body>
<%= form_tag :action=>'authenticate' %>
   <table >
       <tr align="center" class="tablebody">
          <td>User name:</td>
          <td><%= text_field("user", "user_name",:size=>"15" ) %></td>
       </tr>
       <tr align="center" class="tablebody">
          <td>Password:</td>
          <td><%= password_field("user", "password",:size=>"17" )
                                                         %></td>
       </tr>
       <tr align="center" class="tablebody">
          <td></td>
          <td><input type="submit" value=" LOGIN " /></td>
       </tr>
   </table>
</body>
</html>
```

Next, we have to add the link to the stylesheet that we will be using. RoR makes adding stylesheet reference simpler by providing the `stylesheet_link_tag` helper. It takes the following arguments:

- **Name**—It specifies the name of the stylesheet being included.
- **Media**—It specifies on which kind of media the page is being shown. It can be a hand held device such as a cell phone or it can be a normal browser. If you provide 'all' as value, it becomes generic for all the devices.

The name of the stylesheet is 'basic' and the media is 'all'. Before we add the stylesheet tag to the page, let us create the stylesheet. Open your favorite editor and write the following code:

```
BODY {
  background-color: #ffffff;
}
BODY, P, TD, OPTION, SELECT, INPUT, TEXTAREA {
  font-size: .98em;
}
PRE {
  font-size: 0.9em;
}
BODY, P, TD, OPTION, SELECT {
  font-family: Arial, Helvetica, sans-serif;
}

PRE, TEXTAREA {
  font-family: monospace;
}
INPUT {
  font-family: Arial, Helvetica, sans-serif;
}
HR {
}
A {
  text-decoration: underline;
}
A:link {
  color: #0000ff;
}
A:visited {
  color: #0000ff;
}
A:active {
  color: #C01010;
}
```

```
.talewikiheader {
  text-align: right;
}
.talewikifooter {
  text-align: right;
}
```

Save it as `basic.css` in the `public/stylesheets` folder of the talewiki folder. If you observe, the Convention-over-Configuration principle is evident here also. RoR looks for a stylesheet named `basic.css` inside the folder named `public/stylesheet`. Add the stylesheet tag so that the code looks like as follows:

```
<html>
<head>
<%= stylesheet_link_tag "basic", :media => "all" %>
</head>
<body>
<%= form_tag :action=>'authenticate' %>
   <table >
       <tr align="center" class="tablebody">
          <td>User name:</td>
          <td><%= text_field("user", "user_name",:size=>"15" ) %></td>
       </tr>
       <tr align="center" class="tablebody">
          <td>Password:</td>
          <td><%= password_field("user", "password",:size=>"17" )
                                                       %></td>
       </tr>
       <tr align="center" class="tablebody">
          <td></td>
          <td><input type="submit" value=" LOGIN " /></td>
       </tr>
   </table>
</body>
</html>
```

Coming back to the header, add an `<img>` tag enclosed by the `div` tag. Then, change the table and row/column tags so that it looks like as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>TaleWiki</title>
<%= stylesheet_link_tag "basic", :media => "all" %>
</head>
```

```
<body>
<div class="talewikiheader">
<div align="center">
   <img src="/images/talewiki.jpg" alt="[TaleWiki]">
</div>
</div>
<hr>
<%= form_tag :action=>'authenticate' %>
<div>
<center>
<table border=1><tr><td>
<table border=0 cellspacing=0>
   <tr>
      <td colspan=2 class="tabletitle">
         <b>Please enter your user information</b>
      </td>
   </tr>
   <tr>
      <td align="right">
         Username:
      </td>
      <td>
         <%= text_field("user", "user_name",:size=>"15" ) %>
      </td>
   </tr>
   <tr>
      <td align="right">
         Password:
      </td>
      <td>
         <%= password_field("user", "password",:size=>"17" ) %>
      </td>
   </tr>
   <tr>
      <td colspan=2 align="center">
         <HR>
         <input type="submit" value=" LOGIN " />
      </td>
   </tr>
</table>
</td></tr></table>
</center>
</div>
<hr>
```

```
<div class="talewikifooter">
</div>
</body>
</html>
```

Place `talewiki.jpg` in the `images` folder, which is directly under the `talewiki` folder. This image was developed specifically for the header. That completes the layout design for the login page. Let us save the index page as a layout so that we can use it to override the master layout for the index page. There is a small hitch. The layout will override the notices provided through flash. Flash is a message displayed by RoR to convey any information to the user. Do not confuse it with Adobe's flash. So add the following statement after the <hr> before the <div> for footer.

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

It checks whether the `flash` is a note or a notice and shows it accordingly. Next, let us create a master layout and apply it to the rest of the application.

## Defining the Master Layout

We will be performing two tasks while defining the master layout. They are:

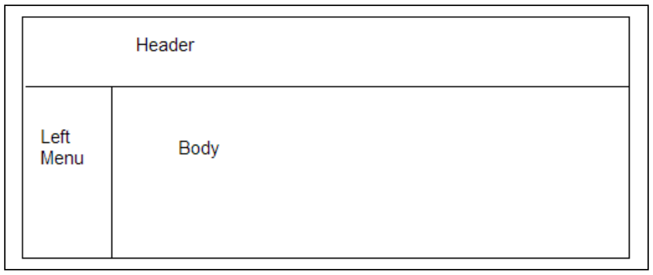
- Defining and applying the Master Layout
- Setting up the Navigation

In the master layout, we will have a placeholder for the menu, which we will replace with the menu when we set up the navigation.

### Defining and Applying the Master Layout

Master layout is an RHTML file just like other layout files, with one difference. The master layout contains structure that will be common to all pages. To insert the content generated by other pages into the master layout, we will use a variable called `@content_for_layout`. This variable is provided by RoR. This variable contains the content/HTML generated by other pages. Wherever RoR sees the `@content_for_layout` variable, it replaces the variable with the content generated by the `action` method that has been called most recently. You can also use the `yield` method instead of the `@content_for_layout` variable. Now let us see `@content_for_layout` in action.

The layout for the pages of TaleWiki will be as follows:



To create the master layout page, open an editor and write the following code:

```
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html;
   charset=windows-1252"/>
<body>
<table width="778" bgcolor="#FFFFFF" height="428" border="0"
cellpadding="0" cellspacing="0">
  <tr>
    <td height="35" width="778" align="left" valign="top">
    <!--header-->
    </td>
  </tr>
  <tr>
    <td width="778">
    <table width="778" border="0" cellpadding="0"
    cellspacing="0">
      <tr>
        <td width="150" align="left" valign="top"
        bgcolor="#FFFFFF">
        <!--leftmenu-->
        </td>
        <td width="478" align="left" valign="top">
        <!--body-->
        </td>
        <td width="150" bgcolor="#FFFFFF" height="384"
                align="left" valign="top">&nbsp;</td>
      </tr>
    </table>
    </td>
  </tr>
</table>
</body>
</html>
```

Save the file as `master.rhtml` in the `app/views/layouts` folder. We will be customizing the highlighted tags, or in terms of the table, we will be adding our content to the highlighted cells. Let us start with the body where the dynamic content will be placed. Replace `<!—body-->` with `<%=@content_for_layout%>` so that the code looks as follows:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;
  charset=windows-1252"/>
<style type="text/css">

body {
  MARGIN: 0px;
}
</style>
<body>
<table width="778" bgcolor="#FFFFFF" height="428" border="0"
cellpadding="0" cellspacing="0">
  <tr>
    <td height="35" width="778" align="left" valign="top">
    <!--header-->
    </td>
  </tr>
  <tr>
    <td width="778">
    <table width="778" border="0" cellpadding="0" cellspacing="0">
      <tr>
        <td width="150" align="left" valign="top" bgcolor="#FFFFFF">
        <!--leftmenu-->
        </td>
        <td width="478" align="left" valign="top">
        <%=@content_for_layout%>
        </td>
      /tr>
    </table>
    </td>
  </tr>
</table>
</body>
</html>
```

Next, let us change the header. Add an `<image>` tag so that it refers to `talewiki.jpg`. The code will look as follows after the addition of the image tag:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=windows-
1252"/>
<style type="text/css">
body {
  MARGIN: 0px;
}
</style>
<body>
<table width="778" bgcolor="#FFFFFF" height="428" border="0"
cellpadding="0" cellspacing="0">
  <tr>
    <td height="35" width="778" align="left" valign="top">
    <img scr= "/images/talewiki.jpg"/>
    </td>
  </tr>
    <tr>
      <td width="778">
      <table width="778" border="0" cellpadding="0" cellspacing="0">
        <tr>
          <td width="150" align="left" valign="top"
          bgcolor="#FFFFFF">
          <!--leftmenu-->
          </td>
          <td width="478" align="left" valign="top">
          <%=@content_for_layout%>
          </td>
          <td width="150" bgcolor="#FFFFFF" height="384" align="left"
          valign="top">&nbsp;</td>
        </tr>
      </table>
      </td>
    </tr>
</table>
</body>
</html>
```

This layout also overrides the flash notices. Therefore, we have to add the code block for showing the flash. Add the following code just before the `<%=@content_for_layout%>` tag.

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

Now, we have to tell RoR to use the master layout. For that, we will have to add the layout declaration to the Controllers. Open the `user_controller.rb` file and place the layout declaration as follows:

```
class UserController < ApplicationController
    before_filter :check_authentic_user, :except => [:index,
    :authenticate]
   layout "master"
:
:
end
```

However, we do not need the master layout for the index page. So, add the `render` declaration in the index method so that the method looks like as follows:

```
def index
    render :layout=>'login'
end
```

The `render` method controls the HTML being generated (also known as page rendering) by a particular method. One of the parameters is `layout`. As, we want to use the login layout, we pass the value `login` to the layout argument. By doing this, we have overridden the master layout for the index method.

Similarly, add the layout declaration to all the other Controllers. That completes defining the master layout step. Now, let us move on to the step of setting up the menu and navigation.

## Setting up the Navigation

Navigation for a site rests solely on two things. They are the menu and internal links between the pages. Menus help the user to jump from one module to another and internal links provide access to the operations within a particular module. In addition, the internal links provide a good mechanism to control the access to certain features. Let us start with setting up the menu.

We will place the menu in the master layout, as we do not require the generation of menus dynamically. We will replace the following comment with the menu:

```
<tr>
  <td width="150" align="left" valign="top" bgcolor="#FFFFFF">
    <!--leftmenu-->
  </td>
```

Open the `master.rhtml` file and replace `<!—leftmenu-->` with the following code:

```
<tr>
  <td width="150" align="left" valign="top" bgcolor="#FFFFFF">
    <table width="778" border="1" cellpadding="0"
    cellspacing="0" height="387">
      <tr><td width="75" height="250">
      <table width="75" border="1" cellpadding="2"
      cellspacing="2">
         <tr>
           <td width="286"><a
           href="http://localhost:3000/role"> Roles</a></td>
         </tr>
           <tr>
             <td><a
             href="http://localhost:3000/user">Users</a></td>
           </tr>
             <tr>
               <td><a
                 href="http://localhost:3000/genre">Genres</a>
               </td>
             </tr>
           <tr>
             <td><a href="http://localhost:3000/tale">
                                        Tales</a></td>
                               </tr>
                               <tr>
                                 <td rowspan="6">&nbsp;</td>
                               </tr>
    </table>
  </td>
```

The menu is very simple. It uses a combination of `<td>`/`<a>` tags to create a menu. The menu items link up with the index page of the different modules except for the user module. The reason is that the index page of the user management module is the login page. I haven't used the `link_to` helper to show that in layouts, simple `<a>` tags will work just fine. You can try out `link_to` as an exercise. Next, we have to ensure that only the users with the administrator role should be able to access the user and role modules. For that we require the role of the currently logged in user. So, add the following statement to the authenticate method:

```
session[:role_name]=valid_user.role.name
```

Now, the method looks as follows:

```
def authenticate
   if request.get?
     render :action=> 'index'
   else
     @user = User.new(params[:user])
     valid_user = @user.check_login
       if valid_user
         session[:user_id]=valid_user.id
         session[:user_id]=valid_user.role.name
         flash[:note]="Welcome "+valid_user.user_name
         redirect_to(:controller=>'tale',:action => "list")
       else
         flash[:notice] = "Invalid User/Password"
         redirect_to :action=> 'index'
       end
   end
end
```

Next, let us surround the user and role menu items with an `if` statement that will check whether the user has a required role or not.

```
<% if session[:role_name]== 'administrator' %>
  <tr>
    <td width="286"><a href="http://localhost:3000/role">
    Roles</a></td>
  </tr>
  <tr>
    <td><a href="http://localhost:3000/user">Users</a></td>
  </tr>
  <tr>
    <td><a href="http://localhost:3000/genre">Genres</a></td>
  </tr>
<%end%>
```

Next, let us prevent users from accidentally deleting or modifying other users' entries. Open `list.rhtml` which is in `app/views/tales`. Surround the `'link_to 'destroy'` and `link_to 'Edit'` tag with an `if` statement that checks the role of the user. It is similar to what we did for the menu items. The code will look like as follows:

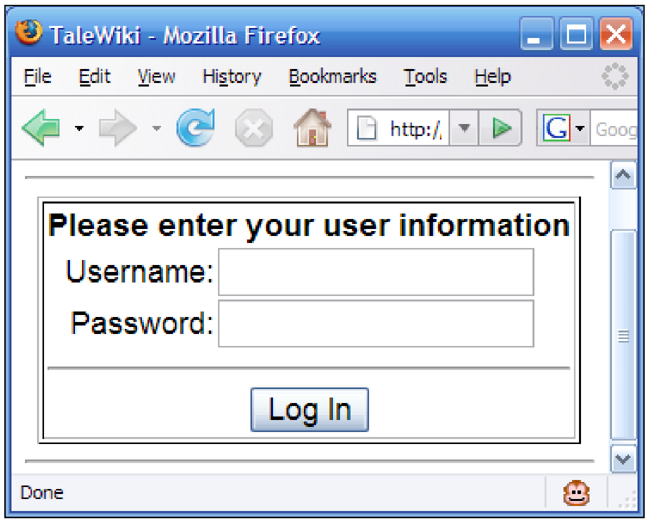
```
<% for tale in @tales %>
  <tr>
  <% for column in Tale.content_columns %>
    <td><%=h tale.send(column.name) %></td>
  <% end %>
    <td><%= link_to 'Show', :action => 'show', :id => tale %></td>
<%if session[:role_name]=='administrator'%>
    <td><%= link_to 'Edit', :action => 'edit', :id => tale %></td>
    <td><%= link_to 'Destroy', { :action => 'destroy', :id =>
    tale }, :confirm => 'Are you sure?', :method => :post
    %></td>
<%end%>
  </tr>
<% end %>
```

That completes setting up the navigation. Next, let us test the modifications we have done.

## Testing the Application

The first page that we are going to test is the login page. Open the following URL in your favorite browser: `http://localhost:3000/user`

If you get the following screen, then the layout for the login page is working fine:


TaleWiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http:/
Goog
Done

Please enter your user information

Username:

Password:

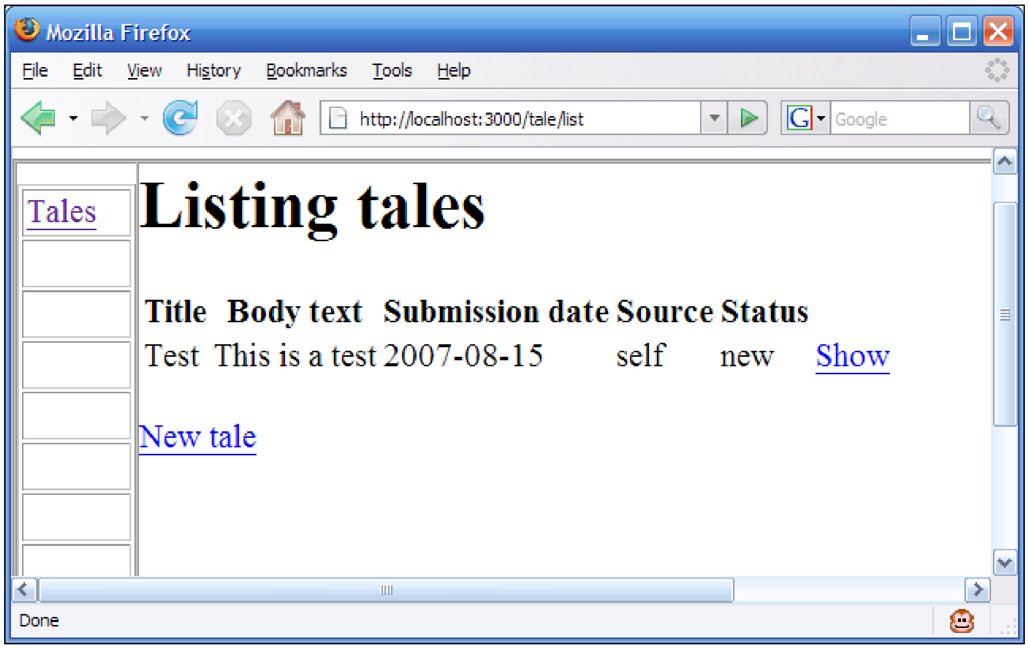
Log In

Now give the following as user name and password

**Username**: **tester**

**Password**: **testing**

If you get the following screen, then the master layout is working fine.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/tale/list
Google
Tales
Done

# Listing tales

| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show |

New tale

That completes the testing of the modifications.

> You can learn more about the layout API from the following URL: `http://api.rubyonrails.org/`.
>
> Look at the documentation of classes under the Action View package for more information on layouts.

## Summary

In this chapter, I discussed about Migrations and layouts. We have also seen how to override the templates whenever required. However, the template that we have set up is very basic. We shall enhance it later when we will implement the personalization. In the next chapter, we will be looking at the implementation of tagging for the tales. So keep on reading.

# 7
# Tagging the Tales

In the previous chapter, we have tackled one aspect of usability (making the site usable for the users), the user interface. Now let us look at the next aspect of usability, which is providing the user with enhanced search within the published tales. Enhancing the search facility is a part of search usability. There are many techniques to provide users with the enhanced search facility. The most used and 'sought after' technique is **Tagging** and **Tag Clouds**. In this chapter, we will be implementing Tagging and Tag Clouds for Talewiki.

The DRY principle is the corner stone of RoR. We have seen this principle in play only in the validation of data. In this chapter, we will be applying the DRY principle using plug-ins to implement the Tagging system. First, we will look at what tagging means and which plug-in are suitable for implementing it. Then we will install the plug-in and set up the tables required by the plug-in. The next step will be implementing the system for Talewiki with the help of the plug-in. We will wrap up the chapter by testing the implemented functionalities.

## Understanding the Requirements

By definition, tags are keywords that classify a set of contents for better searching. So tags are keywords that help in organizing the content according to the user's classification. The organization further helps in searching for the related information without much digging around. Based on the services expected from tags, we can say that the tag management module will provide the following functionalities:

- **Adding a tag**: This operation will provide an interface to add a tag to a particular tale. A tale may have many tags and a tag may be associated with many tales.

- **Search using a tag**: This operation will help the user to find tales that are tagged using the same keyword. The View will be in the form of a list of tales, selecting any one of which will lead to the detailed view of the tale.
- **Visualizing tag clouds**: Tag clouds provide the visualization of the weight of a tag based on its popularity. The most popular tags will be depicted in larger font size or a darker color. The listing will be in alphabetical order.

The next step is to develop the module. So let us get on with the implementation of the Tag Management module.

## Developing the Tag management Module

Until now, we were developing the functionalities from scratch. The reason was that the requirements were TaleWiki-specific. However, the Tag management is totally a different story. Today, tags have become a common tool that aid in searching. Hence, components are available for RoR that provide the basic functionalities required for Tag management. In RoR, components are known as plug-ins. To use a plug-in as the base of our development, the following steps are required:

- Select a plug-in for the Tag management
- Install the plug-in
- Set up tables as required by the plug-in
- Develop the functionalities using the plug-in

Whenever you want to use any plug-in, there may be different options available. Therefore, the first step is a common step when you decide to use a plug-in. Now, let us look at each step in detail.

[ You can find more about the available plug-ins from the following site: `http://agilewebdevelopment.com/plugins`. ]

## Selecting a Plug-in for Tag Management

For Tag Management, one can choose from the following implementations of the Tag management plug-ins:

- **acts_as_taggable**: It was the first plug-in that tried to provide the out-of-the-box implementation for tags. It was developed by D.H.Hansson. The plug-in provided a small set of features. However, tag cloud was not one of them. The status of the plug-in indicates that it has been deprecated and its development has stopped. A new plug-in named as **act_as_taggable_on_steroids** has replaced it.
- **acts_as_taggable_on_steroids**: This plug-in is a rewrite of the act_as_taggable plug-in and can be used in the production environment. Viney Jonathan developed it based on acts_as_taggable. It adds new functionalities for tag implementation including the tag cloud visualization.
- **has_many_polymorphs**: It is a relative newcomer when compared to the other two. One important aspect of this plug-in is that it does not solely focus on the tag implementation. Its focus is on the multi-model joins. In other words, if you require a complex-join query on multiple tables as well as the tag implementation, then this is the one for you.

We can remove the first plug-in—acts_as_taggable—from our list, as it is 'officially' deprecated. From the remaining two, has_many_polymorphs provides functionalities that will not be needed for our development. Therefore, the plug-in that we will be using is acts_as_taggable_on_steroids. From the description of each plug-in, you would also have come to the same conclusion that acts_as_taggable_on_steroids provides the functionalities we need, and it is straightforward as it focuses only on tagging. Let us move onto the next step, which is installing the plug-in.

## Installing the Plug-in

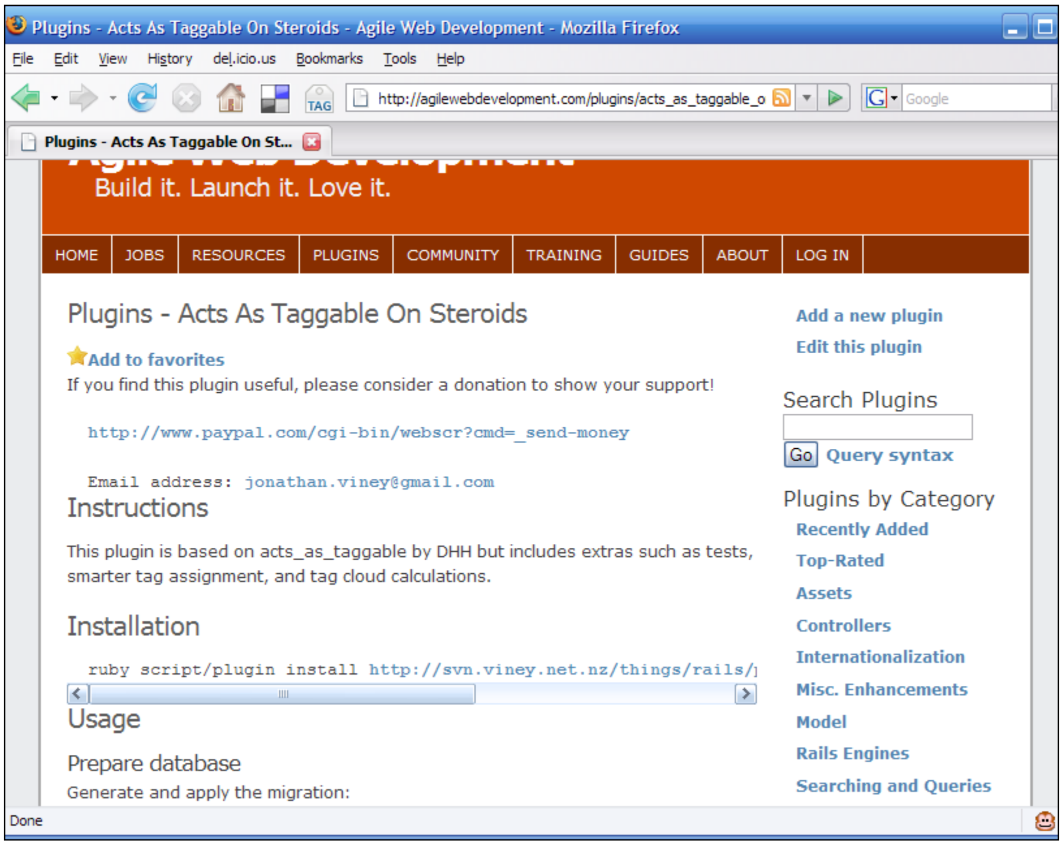
The best place to start looking for the plug-in is the following URL:

`http://agilewebdevelopment.com/plugins/.`

In the search box on the left side, give the following string:

**acts_as_taggable_on_steroids**

Now click **GO**. From the list that the search presents, click on the link for **Acts As Taggable On Steroids**. It will take you to the following page:



Plugins - Acts As Taggable On Steroids - Agile Web Development - Mozilla Firefox

http://agilewebdevelopment.com/plugins/acts_as_taggable_o

Build it. Launch it. Love it.

HOME | JOBS | RESOURCES | PLUGINS | COMMUNITY | TRAINING | GUIDES | ABOUT | LOG IN

Plugins - Acts As Taggable On Steroids

Add to favorites

If you find this plugin useful, please consider a donation to show your support!

```
http://www.paypal.com/cgi-bin/webscr?cmd=_send-money

Email address: jonathan.viney@gmail.com
```

Instructions

This plugin is based on acts_as_taggable by DHH but includes extras such as tests, smarter tag assignment, and tag cloud calculations.

Installation

```
ruby script/plugin install http://svn.viney.net.nz/things/rails/
```

Usage

Prepare database

Generate and apply the migration:

Add a new plugin
Edit this plugin

Search Plugins

Go Query syntax

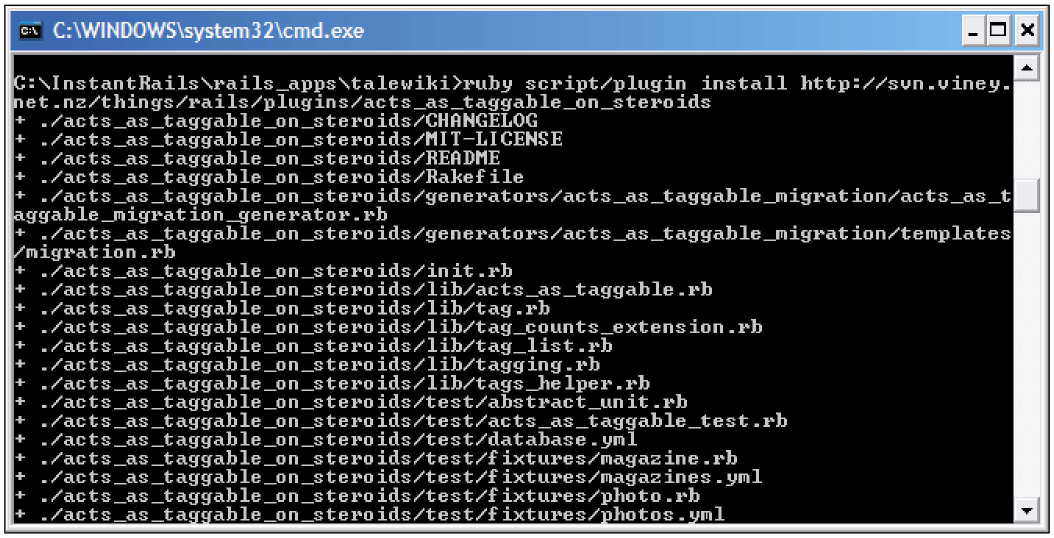
Plugins by Category

Recently Added
Top-Rated
Assets
Controllers
Internationalization
Misc. Enhancements
Model
Rails Engines
Searching and Queries

Now, open the command prompt and give the `use_ruby` command to set up the Ruby and RoR environment. Then, as given in the just shown page, give the following command in the prompt:

```
C:\InstantRails\rails_apps\talewiki>ruby script/plugin install http://
svn.viney.net.nz/things/rails/plugins/acts_as_taggable_on_steroids
```

If you get the following screen, the installation of the plug-in is successful:



```
C:\InstantRails\rails_apps\talewiki>ruby script/plugin install http://svn.viney.
net.nz/things/rails/plugins/acts_as_taggable_on_steroids
+ ./acts_as_taggable_on_steroids/CHANGELOG
+ ./acts_as_taggable_on_steroids/MIT-LICENSE
+ ./acts_as_taggable_on_steroids/README
+ ./acts_as_taggable_on_steroids/Rakefile
+ ./acts_as_taggable_on_steroids/generators/acts_as_taggable_migration/acts_as_t
aggable_migration_generator.rb
+ ./acts_as_taggable_on_steroids/generators/acts_as_taggable_migration/templates
/migration.rb
+ ./acts_as_taggable_on_steroids/init.rb
+ ./acts_as_taggable_on_steroids/lib/acts_as_taggable.rb
+ ./acts_as_taggable_on_steroids/lib/tag.rb
+ ./acts_as_taggable_on_steroids/lib/tag_counts_extension.rb
+ ./acts_as_taggable_on_steroids/lib/tag_list.rb
+ ./acts_as_taggable_on_steroids/lib/tagging.rb
+ ./acts_as_taggable_on_steroids/lib/tags_helper.rb
+ ./acts_as_taggable_on_steroids/test/abstract_unit.rb
+ ./acts_as_taggable_on_steroids/test/acts_as_taggable_test.rb
+ ./acts_as_taggable_on_steroids/test/database.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/magazine.rb
+ ./acts_as_taggable_on_steroids/test/fixtures/magazines.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/photo.rb
+ ./acts_as_taggable_on_steroids/test/fixtures/photos.yml
```

The plug-in command will install the plug-in whose URL has been passed as the argument. That completes the installation step. Next, let us set up the database required by the plug-in.

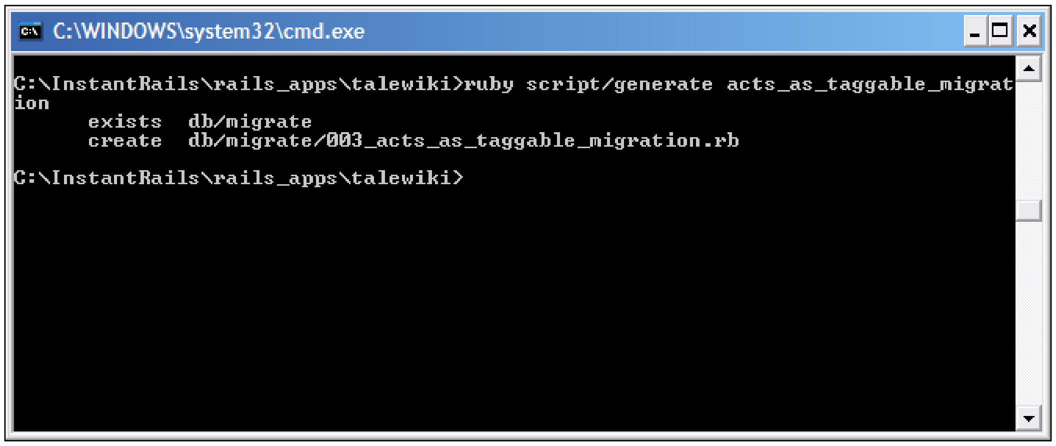
## Setting up Tables Required by the Plug-in

One of the most important aspects of any plug-in is that it abstracts out even the table design. You only have to set up the tables using the migration. It is not a good idea to provide your own tables unless you have a valid reason to do so. In our case, we do not have any reason to provide our own tables. So let us use the tables provided by the plug-in.

The first step in setting up the tables is to generate the migration files. To generate the migration files for the acts_as_taggable_on_steroids plug-in, give the following command in the command prompt:

```
C:\InstantRails\rails_apps\talewiki>ruby script/generate acts_as_
taggable_migration
```

The `generate` command with the `acts_as_taggable_migration` argument will create the migration table in the `db/migrate` directory under the root directory. If you get the following screen, the migration has been generated successfully:

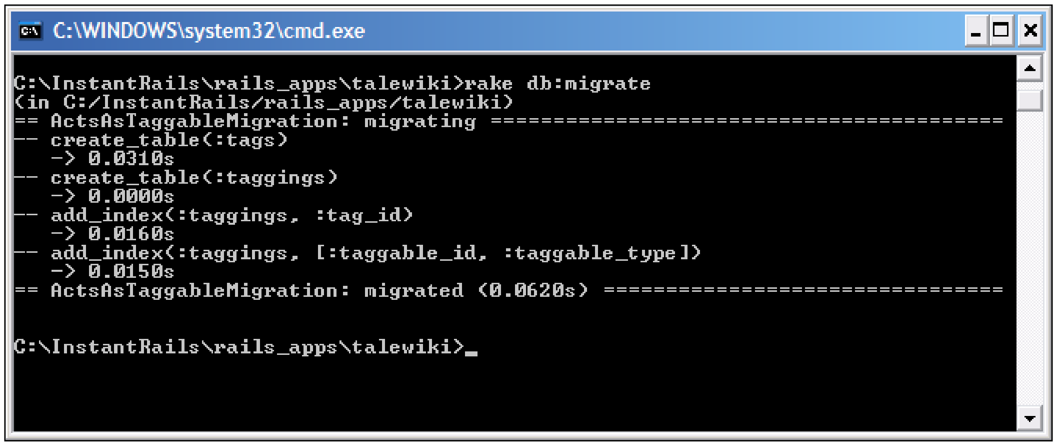


```
C:\InstantRails\rails_apps\talewiki>ruby script/generate acts_as_taggable_migrat
ion
      exists  db/migrate
      create  db/migrate/003_acts_as_taggable_migration.rb

C:\InstantRails\rails_apps\talewiki>
```

Next, let us run the `rake` command on migration so that the tables are created in the database. Give the following command to run the rake tool:

```
C:\InstantRails\rails_apps\talewiki>rake db:migrate
```

If you get the following screen, then the tables have been successfully created:



```
C:\InstantRails\rails_apps\talewiki>rake db:migrate
(in C:/InstantRails/rails_apps/talewiki)
== ActsAsTaggableMigration: migrating ==========================================
-- create_table(:tags)
   -> 0.0310s
-- create_table(:taggings)
   -> 0.0000s
-- add_index(:taggings, :tag_id)
   -> 0.0160s
-- add_index(:taggings, [:taggable_id, :taggable_type])
   -> 0.0150s
== ActsAsTaggableMigration: migrated (0.0620s) =================================


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

The rake tool created two tables and two indexes. The tables are tag and taggings. If you look at the database, you can see these tables. That brings us to the next step—developing the Tag Management modules.

## Developing the Tag Management Module

We are going to develop three functionalities as a part of the Tag Management module. They are:

- Adding a Tag
- Visualizing the Tag cloud
- Searching by Tag

Let us start implementing the functionalities.

## Adding a Tag

This is the first functionality that we will be implementing. The first step is to tell the plug-in about the Model that needs to be tagged. The user will be tagging the Tales. Therefore, we have to tell the plug-in that the Tale model needs to be tagged. To do so, open the `tale.rb` file from the `app/models` folder and add the `acts_as_taggable` declaration to the `Tale` class. After adding the declaration, the `Tale` class will be as follows:

```
class Tale < ActiveRecord::Base
  acts_as_taggable

  validates_presence_of :title, :body_text, :source
  belongs_to:genre
  belongs_to :user
  has_many :comment
end
```

One point to keep in mind is that only the classes derived from `ActiveRecord::Base` will become taggable when `acts_as_taggable` is added. Next, let us provide an interface to the user so that they can add tags to the tales of their choice. Users will be able to add tags when they visit the page having the complete tale (the detailed view of the tales). So let's add a link in the detailed view that will help users to tag that particular tale. Open the `show.rhtml` file which is in the `app/views/tales` folder. The code looks like as follows:

```
<% for column in Tale.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @tale.send(column.name) %>
</p>
<% end %>

<%= link_to 'Edit', :action => 'edit', :id => @tale %>
<%= link_to 'Back', :action => 'list' %>
<%= link_to 'Add Comment',:controller=> 'comment', :action => 'new', :
id => @tale %>
<%= link_to 'View Comments',:controller=>'comment', :action => 'list',
:id => @tale.id %>
```

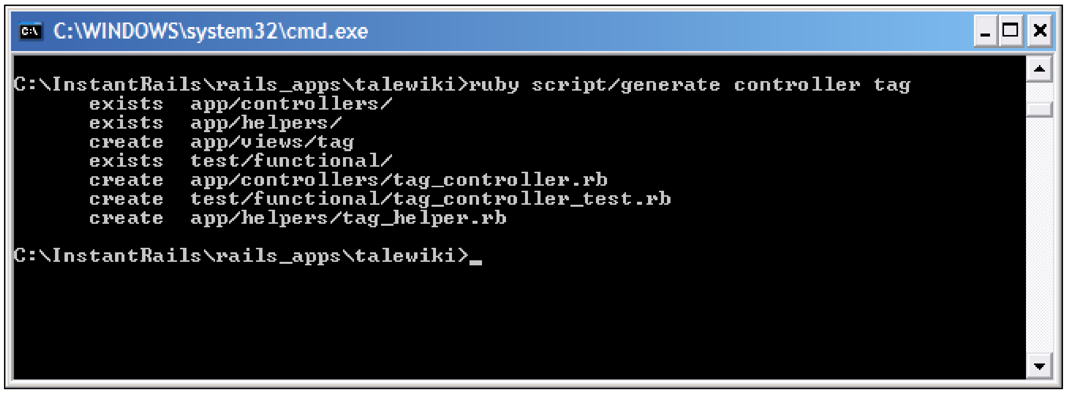
Next, add a link that will take the users to the page where they can add tags. To add the link, add `link_to` helper. The `controller` will be `tag` and the `action` will be `new`. After the modifications, the code will be:

```
<% for column in Tale.content_columns %>
<p>
  <b><%= column.human_name %>:</b>
<%=h @tale.send(column.name) %>
</p>
<% end %>
<%= link_to 'Edit', :action => 'edit', :id => @tale %>
<%= link_to 'Back', :action => 'list' %>
<%= link_to 'Add Comment',:controller=> 'comment', :action => 'new', :
id => @tale %>
<%= link_to 'View Comments',:controller=>'comment', :action => 'list',
:id => @tale.id %>
<%= link_to 'Add tags',:controller=>'tag', :action => 'new', :id => @
tale.id %>
```

The next step is to create the tag controller. In the command prompt, give the following command:

**`C:\InstantRails\rails_apps\talewiki>ruby script/generate controller tag`**

You will get the following screen telling you that the Controller named tag has been created successfully:



```
C:\InstantRails\rails_apps\talewiki>ruby script/generate controller tag
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/tag
      exists  test/functional/
      create  app/controllers/tag_controller.rb
      create  test/functional/tag_controller_test.rb
      create  app/helpers/tag_helper.rb

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

Open the `tag_controller.rb` file within the `app/controllers` folder. You will see the following code:

```
class TagController < ApplicationController
end
```

Next, we have to add the action method given in the `link_to` helper. It is `new`. After adding the method, the `TagController` class will be as follows:

```
class TagController < ApplicationController
  def new
  end
end
```

When the user adds a tag, it will be easier for him/her if he/she sees the title of the tale, which he/she is tagging. To display the title of the tale, we will need the tale object. To get the tale object we will call the find method on the `Tale` class, with the id passed from the show template of the Tale management. Also, we will set the id of the tale in the session so that we can access it easily whenever required.

```
class TagController < ApplicationController
  def new
    @tale=Tale.find(params[:id])
    session[:tale_id]=params[:id]
  end
end
```

Next, we have to create a template that will provide an interface to the user to add tags. To do so, create a file named `new.rhtml` in the `app/views/tag` folder. Open the file in your favorite editor and add the following code:

```
<h1>Add Tag</h1>
<% form_tag :action => 'create' do %>
  <p><label for="tag_tale_title">Title of the Tale</label><br/>
<%=@tale.title%>
<br/>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name' %></p>
<br/>
  <%= submit_tag "Add Tag" %>
<% end %>
<%= link_to 'Back',controller=>'tale' :action => 'list' %>
```

The page calls the `create` method of the tag controller when the submit button is clicked. Therefore, we will be creating the `create` method next. The `create` method will add the tags that the user entered to the tale. To do so, we will need the tale. We will get the tale using the `tale_id` from the session. Then we will call the `tag_list` method on the `tale` object. You will be wondering how we can call a method of the plug-in on an object of `tale`. We are able to do so because of the `acts_as_taggable` declaration in the `Tale` model. Once tagging is done, we will redirect the user to the list of tales. So, here is the `create` method:

```
def create
    tale=Tale.find(session[:tale_id])
    tags=params[:tag]
    tags_tale=tags[:tag_name]
    tale.tag_list=params[:tags_tale]
    if tale.save
      flash[:notice] = 'The tale was successfully tagged'
      redirect_to  controller=>'tale',:action => 'list'
    else
      redirect_to :action => 'new'
    end
end
```

The `tag` parameter returns a hash containing `tag_name` as the key. Therefore, we will have to extract the tags from the hash. That is the reason for the following two statements in the `create` method:

```
tags=params[:tag]
tags_tale=tags[:tag_name]
```

That completes the tagging functionality. Next, let us implement the tag cloud visualization.

## Visualizing the Tag Cloud

Tag cloud visualization is a technique using which the most searched tags can be displayed with a different color or size. The `acts_as_taggable_on_steroids` provides a handy method to achieve the tag cloud visualization. Let us implement the tag cloud visualization for TaleWiki. Open `standard.rhtml` from the `app/views/layouts` folder. The code for the left menu is as follows:

```
<table width="778" border="1" cellpadding="0" cellspacing="0"
height="387">
   <tr><td width="75" height="250"><table width="75"
   border="1" cellpadding="2" cellspacing="2">
      <%if session[:role]=='administrator' %>
```

```
          <tr>
            <td width="286"><a
              href="http://localhost:3000/role"> Roles</a></td>
          </tr>
          <tr>
             <td><a
           href="http://localhost:3000/user/list">Users</a></td>
           </tr>
          <tr>
            <td><a
            href="http://localhost:3000/genre">Genres</a></td>
          </tr>
          <%end%>
          <tr>
            <td><a
            href="http://localhost:3000/tale">Tales</a></td>
          </tr>
          <tr>
            <td rowspan="6">&nbsp;</td>
          </tr>
  </table>
```

Let us add a link labelled `Browse By Tags` that will take the user to the page with the tag cloud. The controller is tag and the action is browse.

```
  <table width="778" border="1" cellpadding="0" cellspacing="0"
  height="387">
     <tr><td width="75" height="250"><table width="75"
     border="1" cellpadding="2" cellspacing="2">
        <%if session[:role]=='administrator' %>
           <tr>
              <td width="286"><a
              href="http://localhost:3000/role"> Roles</a></td>
           </tr>
           <tr>
              <td><a
           href="http://localhost:3000/user/list">Users</a></td>
           </tr>
           <tr>
              <td><a
              href="http://localhost:3000/genre">Genres</a></td>
           </tr>
           <%end%>
           <tr>
              <td><a
```

```
               href="http://localhost:3000/tale">Tales</a></td>
            </tr>
            <tr>
               <td><a
               href="http://localhost:3000/tag/tag_clouds">Browse
               By Tags</a></td>
            </tr>
            <tr>
            <td rowspan="6">&nbsp;</td>
            </tr>
</table>
```

Next, open the `tag_controller.rb` file and add the `tag_clouds` method to the class. We will use the `tag_counts` method to get the tag count. The method will be as follows:

```
def tag_clouds
   @tags=Tale.tag_counts
end
```

The next step in visualization is to call the helper provided by the plug-in from within our helper class. You will find the file of the helper class in `app/helpers`. We will be calling the helper provided by the plug-in in `application_helper.rb`. Any helper method provided in the `AplicationHelper` class will be available throughout the application. To use the helper method provided by the `acts_as_taggable_on_steroids` plug-in, we have to use the `include` declaration. The argument to be included will be the helper. In our case, the name of the helper is `TagsHelper`. After adding the declaration, the `ApplicationHelper` class will look like as follows:

```
module ApplicationHelper
include TagsHelper
end
```

Next, let us implement the view. Create a new RHTML template with name `tag_cloud.rhtml` in the `app/views/tag` folder. Then add the following code:

```
<html>
<head>
<style>
 .cssSmall { font-size: 1.0em; }
  .cssMedium { font-size: 1.2em; }
  .cssNormal { font-size: 1.4em; }
  .cssLarge { font-size: 1.6em; }
</style>
</head>
< body/>
</html>
```

We will be using the style with the tag helper. Next, in the body part, we will call the `tag_cloud`. The name is that of the controller. The helper works by passing the required data back to the controller. The code is as follows:

```
<html>
<head>
<style>
 .cssSmall { font-size: 1.0em; }
  .cssMedium { font-size: 1.2em; }
  .cssNormal { font-size: 1.4em; }
  .cssLarge { font-size: 1.6em; }
</style>
</head>
<body>
<% tag_cloud @tags, %w(cssSmall cssMedium cssNormal cssLarge) do |tag,
css_class| %>
    <%= link_to tag.name, { :action => "result_search", :id => tag.
name }, :class => css_class %>
<% end %>
</body>
</html>
```

The `tag_cloud` helper selects the count array and the style. Then it applies the style according to the weight of the individual count. If you observe, the Convention-over-Configuration is still at play here. Even though the name of the helper and that of the action method are same, yet, as the name is not given as an argument to the action key of any hash, it is treated as a helper. In the `link_to` helper, we are passing an action method called `result_search`. We will be coming to this function in the next section.

That completes the functionality for tag cloud visualization. Next, let us implement the search by tag functionality.

## Searching By Tag

As each tale is tagged, we can provide interface to the user to search by tag. The first step is to provide a link to the search page. Open `master.rhtml` and add a link labelled `Search By Tag` to the left menu. The controller will be `tag` and the action method will be `search`. After addition of the link the left menu will be as follows:

```
<table width="778" border="1" cellpadding="0" cellspacing="0"
height="387">
   <tr><td width="75" height="250"><table width="75"
   border="1" cellpadding="2" cellspacing="2">
     <%if session[:role]=='administrator' %>
```

```
          <tr>
            <td width="286"><a
            href="http://localhost:3000/role"> Roles</a></td>
          </tr>
          <tr>
            <td><a
            href="http://localhost:3000/user/list">Users</a></td>
          </tr>
          <tr>
            <td><a
            href="http://localhost:3000/genre">Genres</a></td>
          </tr>
        <%end%>
          <tr>
            <td><a
            href="http://localhost:3000/tale">Tales</a></td>
          </tr>
          <tr>
            <td><a
            href="http://localhost:3000/tag/tag_cloud">Browse By
            Tags</a></td>
          </tr>
          <tr>
            <td><a href="http://localhost:3000/tag/search">Search
            By Tag</a></td>
          </tr>
:
:
</table>
```

Next, open `tag_controller.rb` and add the `search` method to it. It will be an empty method.

```
def search
end
```

After that, create an RHTML template with the name `search.rhtml`. It will contain the code to show the search box to the user and a button to submit the query. The page will submit the query to the `result` method. Here is the code:

```
<h1>Search By Tag</h1>
<% form_tag :action => 'result' do %>
<p>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name'  %></p>
```

```
<br/>
  <%= submit_tag "Search" %>
<% end %>
<%= link_to 'Back',controller=>'tale' :action => 'list' %>
```

Now, we will use the `find_tagged_with` method to search for the tales with the tag provided by the user. The `find_tagged_with` method takes a comma-delimited string as the argument. Therefore, the `result` method will be as follows:

```
def result
    tags=params[:tag]
    query_tag=tags[:tag_name]
    @tales =Tale.find_tagged_with(query_tag)
end
```

Next, create another template related to the `result` method named `result.rhtml` in `app/views/tag`.

```
<h1>Search Result</h1>
<table>
  <tr>
  <% for column in Tale.content_columns %>
    <th><%= column.human_name %></th>
  <% end %>
  </tr>
<% for tale in @tales %>
  <tr>
  <% for column in Tale.content_columns %>
    <td><%=h tale.send(column.name) %></td>
  <% end %>
    <td><%= link_to 'Show', :action => 'show', :id => tale %></td>
  </tr>
<% end %>
</table>
```

It displays the columns using the `content_columns` getter of the `Tale` class. The names are then converted to a human readable format. Inside the `for` loop, the result of the search is displayed. That is how `Search By Tag` works. Now let us implement the `result_search` method called by the `tag_cloud` helper. It is like the `result` method. It calls the `find_tagged_with` method. The method looks like as follows:

```
def result_search
   query_tag=params[:tag]
   @tales =Tale.find_tagged_with(query_tag)
end
```

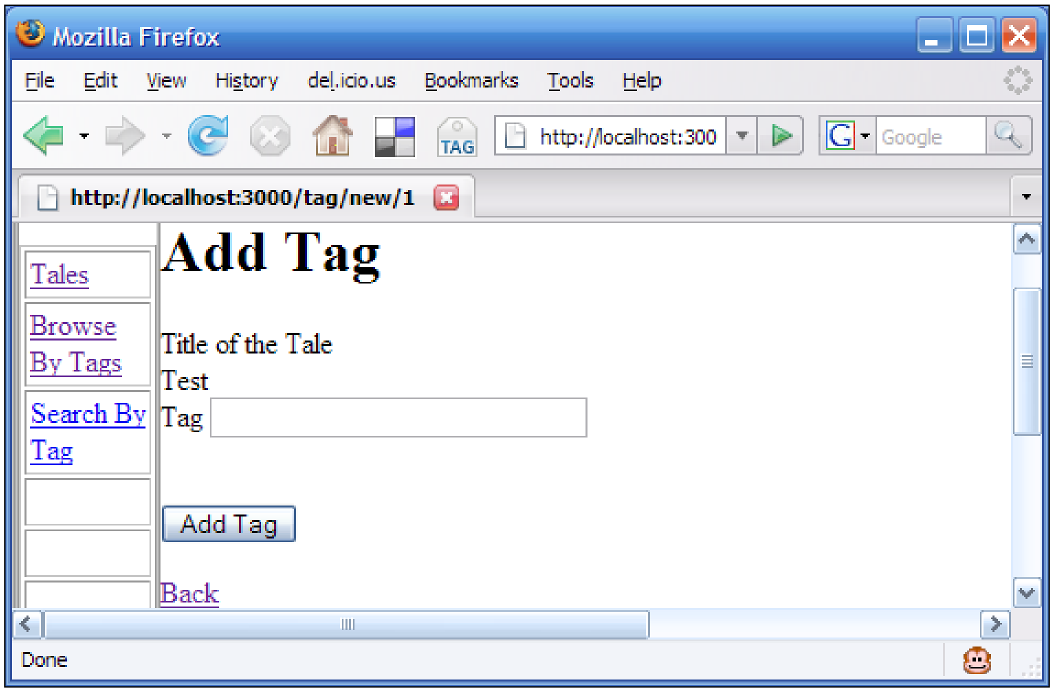
In addition, with that, the development phase of the Tag management module is complete. The only task left is testing the modifications we did.

## Testing the Modifications

Now, let us test the modifications that we have done. First, let us test the 'Add Tag' functionality. Open the following URL:

`http://localhost:3000/tale`.

Click on the **show** link. In the next page, click on the link for **Add Tag**. You will get the following page:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:300
Google
http://localhost:3000/tag/new/1
Tales
Browse By Tags
Search By Tag
Done

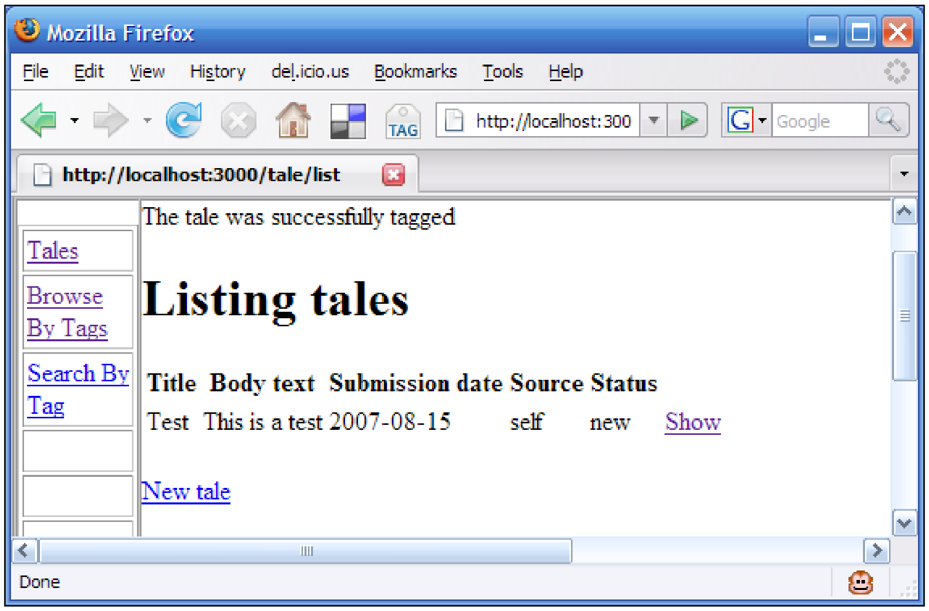
# Add Tag

Title of the Tale
Test
Tag

Add Tag

Back

Give the Tag as **News** and click on **submit**. If you get the following page, then the tag was added successfully:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:300
Google
http://localhost:3000/tale/list
Tales
Browse By Tags
Search By Tag
Done

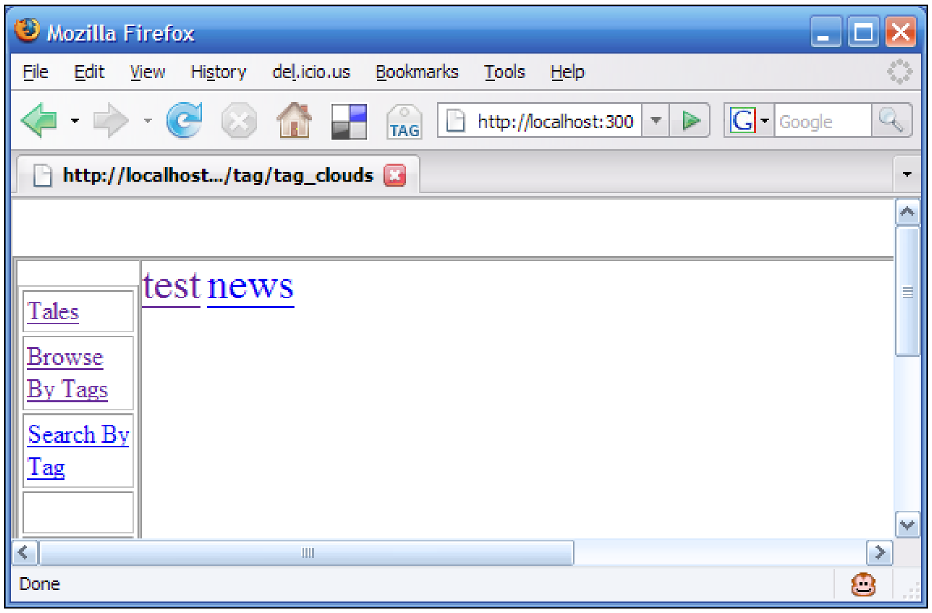
The tale was successfully tagged

Listing tales

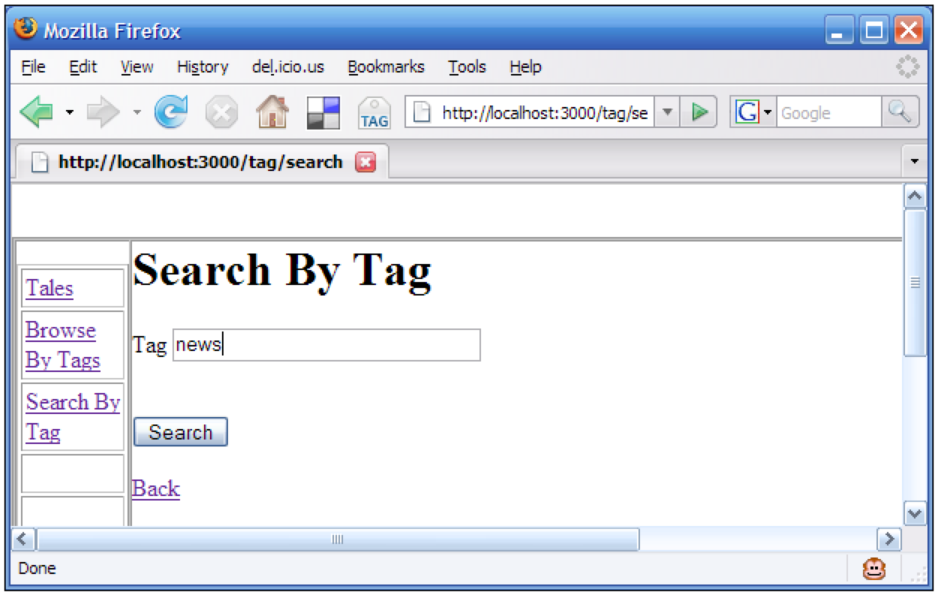
| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show |

New tale

Next, click on  the **Browse By Tags** link of the left menu. If you get the following page, then the tag-cloud visualization is working fine:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:300
Google
http://localhost.../tag/tag_clouds
test news
Tales
Browse By Tags
Search By Tag
Done

Now, click on **Search By Tag** link. You will get the following page:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/tag/se
Google
http://localhost:3000/tag/search
Tales
Browse By Tags
Search By Tag
Done

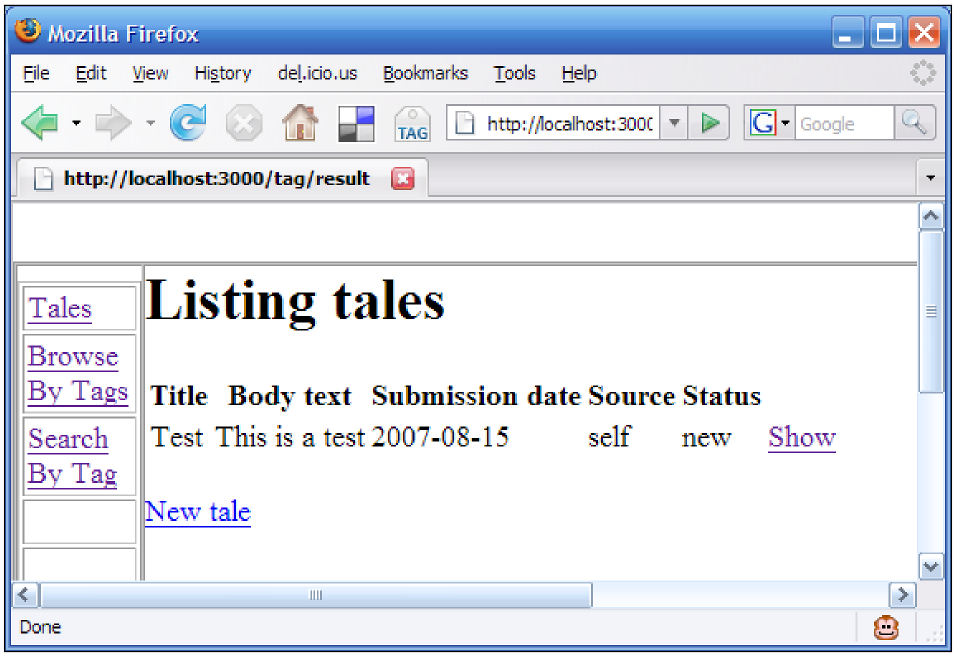
Search By Tag

Tag news

Search

Back

Give **news** as the **Tag** for searching and click **Search**. If you get a page similar to the following page, then the search functionality is working fine.


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:300(
Google
http://localhost:3000/tag/result
Tales
Browse By Tags
Search By Tag
Done

Listing tales

```
Title Body text  Submission date Source Status
Test  This is a test 2007-08-15  self   new    Show
```

New tale

That completes the testing part.

## Summary

In this chapter, we saw how easy it is to use the plug-ins and to implement the functionalities that we need using the plug-ins. However, the functionalities of adding a tag and many others of its kind do a page refresh. In the next chapter, we will see how to overcome this limitation. So keep reading!

# 8
# Enhancing User Experience with Ajax

In the last chapter, we saw how to enhance the search usability and the user experience of TaleWiki by making use of tags and providing users with the tag-based search facility. Searching and tagging are just two of the many aspects of user experience. The third aspect that we need to take care of consists of **interactivity** and **responsiveness**. Interactivity ensures that the users have constant interaction with the TaleWiki, and responsiveness will help in providing results to the user with minimum waiting. In web applications, Ajax is the way to achieve the same thing. In this chapter, we will see how to enhance the various functionalities that we have implemented until now, in order to enhance the usability. First, we will find out the functionalities that can be enhanced using Ajax, then we will look at how RoR supports Ajax, and will be winding up the chapter by implementing the enhancements.

## Understanding the Requirements

The important point to keep in mind while 'Ajaxifying' any functionality is that, not all functionalities will get more responsive and interactive by using Ajax in them. For example, if you want the user to be able to bookmark a page, then its better not to Ajaxify that page. The reason is that the Ajaxified page is not registered. Due to such situations, we need to be careful while choosing the functionalities for 'Ajaxifying'. Keeping this point in mind, let us see on which functionalities we can apply Ajax. The criteria on which we can decide this is the 'number of clicks required to access the functionality'. On the basis of this criterion, the following are the functionalities that we can Ajaxify:

- **Live search**: The search by tag functionality is static. That means, until the user clicks on submit after entering the tag to be searched, the search results will not be displayed. Using Ajax, we can make it live. That means, as the user types the search string, the results will be displayed. In addition, in every 0.25 seconds, we will check whether a new string has been entered. If it has been entered, we will then display the results accordingly.
- **Editing functionality**: The edit/modification functionality that we have developed takes the user to an edit page. Using Ajax, we will provide in-line editing. By using in-line editing, the user will not need to navigate to another page to edit or update the information such as their password.

We are going to Ajaxify these functionalities. Now let us start working on Ajaxifying the functionalities.

## Implementing Ajax

We have defined the functionalities that we are going to Ajaxify. However, before we implement Ajax, we need to understand two points:

- What is Ajax
- How RoR and Ajax are related

Let us have a look at these points.

### What is Ajax?

Ajax is not a single technology. It is, essentially, a combination of technologies. The term Ajax can be expanded to **Asynchronous** JavaScript and XML. The keyword here is Asynchronous. Typically, a web page will stop all its work until it receives a response from the server. For such a web page, the communication (sending a request to the server and getting a response from the server) is synchronous. However, if a web page sends the requests and does not wait for the reply to resume processing, then it uses the asynchronous method of communication. Ajax helps developers to implement the asynchronous method of communication using JavaScript.

The `XMLHTTPRequest` object of JavaScript can be used to send a request to the server and then do the required changes to the web page based on the data sent by the server. Now let us look at the data sent by the server. The server sends its response as an XML document. It is from XML that Ajax gets its 'X.' At the client side you can use the JavaScript XML parsing API to parse the XML and get the data. This data can be used to modify the page or display the information to the user, using JavaScript.

The problem with JavaScript is that it is browser-dependent. The browser dependency makes it harder to implement Ajax. However, you can use different JavaScript libraries that provide an out-of-the-box implementation for Ajax. That means you do not need to write code to get all the facilities of Ajax. The following are the two most widely used JavaScript libraries having Ajax support:

- **Prototype**: It is the JavaScript library/framework developed by Sam Stephson, which provides a simplified Ajax API and other utilities, such as accessing form variables, replacing data within a set of tags, and so on.
- **Script.aculo.us**: It is pronounced as 'Scriptaculous'. This library/framework is built upon *prototype* to provide dynamic visual effects. The effects include animation of controls and GUI elements, fading, and so on.

That was a 'bird's eye' view of Ajax. You do not need to worry about adding JavaScript to your code. The reason is that Ajax and other JavaScript functionalities are present as the core functionalities of RoR. How does that translate to the ease of development? That is what we are going to see next.

## How Ajax and RoR are Related

The DRY principle of RoR is evident in the approach that it uses towards Ajax and other JavaScript related functionalities. RoR has integrated various JavaScript libraries/frameworks into its core. RoR provides these libraries to the developer as JavaScript helpers. RoR provides four libraries out-of-the-box to the users. They are:

- **Prototype**: This library is one of the main JavaScript libraries that RoR supports. The development of prototype is driven by its integration with RoR.
- **Effects**: The Script.aculo.us library is divided into different helpers. The `Effects` helper is one of them. It provides visual effects for the GUI elements and controls.
- **DragDrop**: The 'drag and drop' functionalities that Script.aculo.us gives are provided by the DragDrop helper. Using this helper, you can develop a 'drag and drop' based GUI that can be customized by the user.
- **Controls**: This library provides the Ajax-based controls. Using these, you can provide data-tables that update automatically on the basis of the user's selection.

The next obvious question will be how to use these helpers within a page. To use these helpers, we will have to use the `javascript_include_tag`, which is one of the helpers that the `JavaScriptHelper` module provides. Using the `javascript_include_tag` will import the JavaScript library provided as a part of the argument. The `javascript_include_tag` accepts the following argument:

- **Name of the JavaScript library**: This is the only argument that the `javascript_include_tag` accepts. The argument is the name of the library or the path to the JavaScript that you want to use. The libraries you can choose are any one of the previously mentioned four libraries. The most commonly passed argument is `:defaults` which includes all the libraries.

Now that we have familiarized ourselves with the way of Ajax in RoR, let us move on to the implementation of the functionalities. First, we will implement the live search functionality.

## Implementing the Live Search

To convert a simple search to a live search, we will be following these steps:

- Specify the location to display the result
- Use the `observe_field` helper to monitor the search field
- Modify the action method

Let us see them in detail.

### Specify the Location to Display the Result

First, we have to tell RoR where we will display the results. To do so, we will use the `<div>` tag. Before that, we will need to do some modifications to the 'Search by Tag' page. Open `search.rhtml` from the `app/views/tag` folder. The current code looks like as follows:

```
<h1>Search By Tag</h1>
<% form_tag :action => 'result' do %>
<p>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name'  %></p>
<br/>
  <%= submit_tag "Search" %>
<% end %>
<%= link_to 'Back',:controller=>'tale',:action => 'list' %>
```

First, we will remove the `submit` button so that the user does not click it. Along with the submit button, we will remove the `form_tag` helper because there is no requirement for the form tag. After the modifications, the code will be as follows:

```
<h1>Search By Tag</h1>
<p>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name'  %></p>
<br/>
<%= link_to 'Back',:controller=>'tale',:action => 'list' %>
```

Next, let us specify where the result will be displayed. It will be before the `link_to 'Back'` statement and after the `<br/>` tag. It is specified using the `<div>` tag. It will act as the place holder for the content. Let us name it `result`. After this, the code will be as follows:

```
<h1>Search By Tag</h1>
<p>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name'  %></p>
<br/>
  <div id="result"></div>
<%= link_to 'Back',:controller=>'tale',:action => 'list' %>
```

That completes the first step. Next, let us get the results using the `observe_field` helper.

## Use the observe_field Helper

The next step is to use the `observe_field` helper. In order to use it, we will need the JavaScript libraries. To include the JavaScript libraries, we will use the `javascript_include_tag` helper. Placing the JavaScript in the head part is a good practice because the head part will be evaluated before any other tags. Hence, errors such as the **No method found** error can be avoided when the functions of JavaScript are called later on. We will use `:defaults` as the parameter. The code after adding the `javascript_include_tag` helper will be as follows:

```
<html>
<head>
<%= javascript_include_tag :defaults %>
</head>
<body>
<h1>Search By Tag</h1>
```

```
<p>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name'  %></p>
<br/>
  <div id="result"></div>
<%= link_to 'Back',:controller=>'tale',:action => 'list' %>
</body>
</html>
```

Next, we will add the `observe_field` helper. It takes the following arguments:

- **Field to observe**: This argument takes the name of the control, which needs to be monitored for change in its value. In our case, it will be `tag_name`.
- **Frequency**: The frequency at which the field will be checked for change in its value is passed as the value for this argument. We will be checking for change in the value every quarter of a second. So, the value we will pass will be 0.25.
- **Update**: This argument takes the element or the tag to be updated with the result as value.  We will be displaying the result in the `<div>` tag whose name is result. Therefore, we will pass `result` as the value to this argument.
- **url**: The `action` method that will provide the result is passed as the value of this argument. In our case, the action method is `result`. So the value to the URL will be `result`.

After adding the `observe_field` helper, the code will be as follows:

```
<html>
<head>
<%= javascript_include_tag :defaults %>
</head>
<body>
<h1>Search By Tag</h1>
<p>
<label for="tag_name">Tag</label>
<%= text_field 'tag', 'tag_name'  %></p>
<br/>
  <%= observe_field(:tag_name,:frequency => 0.25,:update =>
  :search_result,:url => { :action => :result }) %>
  <p>Search Results </p>
  <div id="search_result"></div>
<%= link_to 'Back',:controller=>'tale',:action => 'list' %>
</body>
</html>
```

## Modify the Action Method

The action method passed to `observe_field` renders the result using the master layout. That means, the result that will be displayed will contain the left menu which is undesirable. So we will have to override the layout template. So, create a new template in `app/views/layouts` and name it `empty.rhtml`. It will have the following code:

```
<html>
<head/>
<body>
<%=@content_for_layout%>
</body>
</html>
```

Next, add the statement to override the default template. After the modification, the code will be as follows:

```
def result
   tags=params[:tag]
   query_tag=tags[:tag_name]
   @tales =Tale.find_tagged_with(query_tag)
   render :layout=>"empty"
end
```

That completes the implementation of the live search functionality. Next, we will be working on in-line editing functionality.

## Implementing the In-line Editing

In-line editing, also known as in-place editing, is a way to enhance the usability by providing the modification of the displayed data in the same place without leaving the page. Ajax helped to bring this feature of desktop applications to web applications. To implement in TaleWiki, we will use the `in_place_editor_field` helper. To use `in_place_editor_field` helper we will follow the following steps:

- Mark the fields for in-line editing
- Set up the controller to update the values

I will be implementing the in-line editing for the 'Modify User' operation. The administrator may need to reset the password. Implementing the same thing for the other modules is left to you as an exercise.

## Marking the Fields for In-line Editing

To mark the fields, we will start with modifying `show.rhtml` in the `app/views/users` folder. The `show.rhtml` provides details about a particular user. The in-line edit functionality will come in handy here because the administrator will not be required to go to the edit page just to edit one or two values. The code looks like as follows:

```
<% for column in User.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @user.send(column.name) %>
</p>
<% end %>
<%= link_to 'Edit', :action => 'edit', :id => @user %> |
<%= link_to 'Back', :action => 'list' %>
```

Let us change it so that the fields are shown column-wise rather than row-wise. After the changes, the code will be as follows:

```
<table width="100%" border="1">
  <tr>
    <th scope="row"align="left"> User Name </th>
    <td width="71"><%=@user.user_name%></td>
  </tr>
  <tr>
    <th scope="row" align="left">Password</th>
    <td><%=@user.password></td>
  </tr>
  <tr>
    <th scope="row" align="left">First Name </th>
    <td><%=@user.first_name%></td>
  </tr>
  <tr>
    <th scope="row" align="left">Last Name </th>
    <td><%=@user.last_name%></td>
  </tr>
  <tr>
    <th scope="row" align="left">Age</th>
    <td><%=@user.age%></td>
  </tr>
  <tr>
    <th scope="row" align="left">eMail</th>
    <td><%=@user.email%></td>
  </tr>
  <tr>
```

```
      <th scope="row" align="left">Country</th>
      <td><%=@user.country%></td>
    </tr>
    <tr>
      <th scope="row" align="left">&nbsp;</th>
      <td>&nbsp;</td>
    </tr>
  </table>
  <%= link_to 'Edit', :action => 'edit', :id => @user %>
  <%= link_to 'Back', :action => 'list' %>
```

Next, let us add the JavaScript helper tag as we will be using the Script.aculo.us library. As the Script.aculo.us library requires the prototype library, we will use `:defaults` as the argument. After adding the helper, the code will be as follows:

```
  <%=javascript_include_tag :defaults %>
  <table width="100%" border="1">
    <tr>
      <th scope="row"> User Name </th>
      <td width="71"><%=@user.user_name%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Password</th>
      <td><%=@user.password></td>
    </tr>
    <tr>
      <th scope="row" align="left">First Name </th>
      <td><%=@user.first_name%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Last Name </th>
      <td><%=@user.last_name%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Age</th>
      <td><%=@user.age%></td>
    </tr>
    <tr>
      <th scope="row" align="left">eMail</th>
      <td><%=@user.email%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Country</th>
      <td><%=@user.country%></td>
    </tr>
    <tr>
      <th scope="row" align="left">&nbsp;</th>
      <td>&nbsp;</td>
    </tr>
  </table>
  <%= link_to 'Edit', :action => 'edit', :id => @user %>
  <%= link_to 'Back', :action => 'list' %>
```

Next, let us mark the fields for in-line editing. To do so, we will use the `in_place_editor_field` helper. It accepts the following arguments:

- **:fields** : It specifies the fields that require the in-place editing.
- **:options**: The highlighting color, the size of the field, are passed as options. The `options` parameter takes hash as its value.
- **:url**: The action to be called when the user clicks on the **OK** button. The `url` is passed as a hash containing the controller and the action to be called as values. If you are going to use the helper to update values, then this argument is not optional.

Therefore, after adding the `in_place_editor_field`, the code will be as follows:

```
  <%=javascript_include_tag :defaults %>
  <table width="100%" border="1">
    <tr>
      <th scope="row" align="left"> User Name </th>
      <td width="71"><%=@user.user_name%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Password</th>
      <td><%=in_place_editor_field('user','password' )%></td>
    </tr>
    <tr>
      <th scope="row" align="left">First Name </th>
      <td><%=@user.first_name%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Last Name </th>
      <td><%=@user.last_name%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Age</th>
      <td><%=@user.age%></td>
    </tr>
    <tr>
```

```
      <th scope="row" align="left">eMail</th>
      <td><%=@user.email%></td>
    </tr>
    <tr>
      <th scope="row" align="left">Country</th>
      <td><%=@user.country%></td>
    </tr>
    <tr>
      <th scope="row" align="left">&nbsp;</th>
      <td>&nbsp;</td>
    </tr>
  </table>
  <%= link_to 'Edit', :action => 'edit', :id => @user %>
  <%= link_to 'Back', :action => 'list' %>
```

We will provide the facility to change only the password, unless the user/administrator goes for a complete modification using the **Edit** link. This will help the administrator to reset the password. Next, let us do the changes in the controller.

## Set up the Controller

The `in_place_edit_field` helper has a corresponding helper for the controller. It is the `in_place_edit_field_for` helper. It takes the following arguments:

- **Name of the model**: The model that will be updated will be the value for it. In our case, it is user.
- **Name of the field:** The attribute of the model (column of the table) that will be updated becomes the value of this argument. In our case, it is password.

Now, let us add the helper to the **user_controller.rb**. Add the `in_place_edit_field_for` helper after the filter declaration. After the modification, `user_controller.rb` will be as follows:

```
class UserController < ApplicationController
 layout "master"
 before_filter :check_authentic_user, :except => [:index, :
authenticate]
 in_place_edit_field_for :user, :password
#the remaining code
end
```

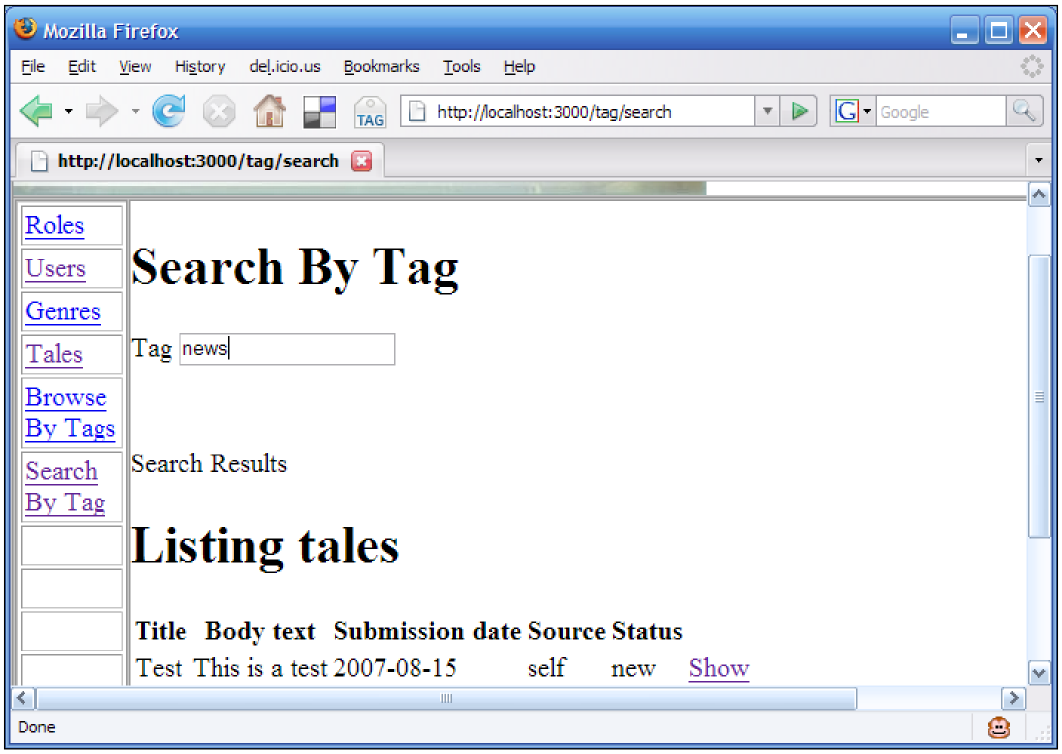
That completes the implementation of in-line editing functionality. Now, let us test the modifications.

## Testing the Modifications

Let us first test the live-search functionality. Open your favorite browser and give the following URL:

`http://localhost:3000/user`

After logging in using the administrator as user name and admin as password, click on the **Search by Tag** link. In the search field, give **news** as input. If you get the following screen, then the modifications are working fine:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/tag/search
Google
Roles
Users
Genres
Tales
Browse By Tags
Search By Tag
Done

Search By Tag

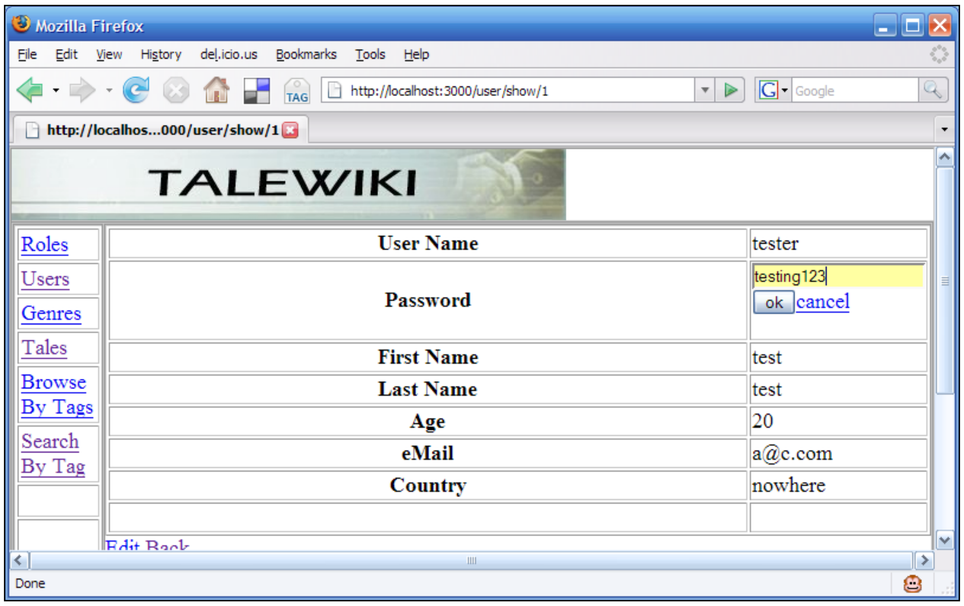
Tag news

Search Results

Listing tales

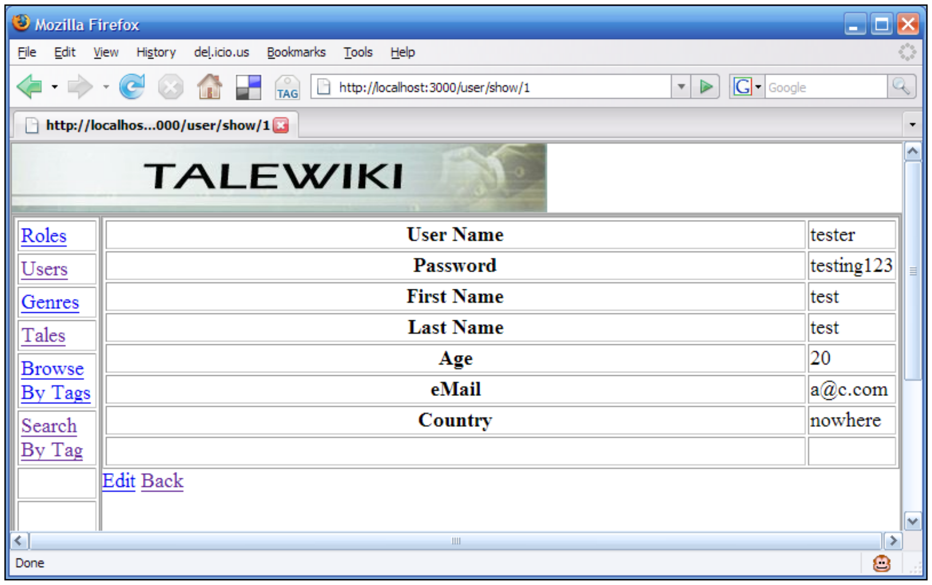
| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show |

Next, click on the **Users** link in the left menu. Then click on the **Show** link of user with **User Name** as **tester**. On the next page, click on the value displayed against the **Password** field. If you get the following page, then everything is fine:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/user/show/1
http://localhos...000/user/show/1
TALEWIKI
Roles
Users
Genres
Tales
Browse By Tags
Search By Tag
ok
cancel
Edit Back
Done

| | |
|---|---|
| User Name | tester |
| Password | testing123 |
| First Name | test |
| Last Name | test |
| Age | 20 |
| eMail | a@c.com |
| Country | nowhere |

Change the password and click on the **OK** button. If you get the following page, then the modifications are working fine:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/user/show/1
http://localhos...000/user/show/1
TALEWIKI
Roles
Users
Genres
Tales
Browse By Tags
Search By Tag
Edit Back
Done

| | |
|---|---|
| User Name | tester |
| Password | testing123 |
| First Name | test |
| Last Name | test |
| Age | 20 |
| eMail | a@c.com |
| Country | nowhere |

That completes the testing of our modifications.

## Summary

That completes implementing the Ajax for TaleWiki. This is the second step towards enhancing the usability. There are many more steps in the process of enhancing usability, which we will see later. In the next chapter, we will implement the administrative interface so that the administrator can efficiently manage the website. So keep reading!

# 9 Developing the Interface for Administration

In the previous chapters, we have specified certain functionalities as accessible only to the administrator. However, we still haven't developed a User Interface from where the administrator can access all of them without navigating between multiple modules. In this chapter, we will be designing such an interface that will facilitate the administrator in 'administering' the TaleWiki.

First, we will identify from the already implemented functionalities those that an administrator will require to manage the TaleWiki as well as add new ones. Then we will implement them and wrap up the chapter by testing the administrative interface.

## Understanding the Requirements

We have already developed most of the functionalities of the Administration module. Genre management, Role management, and User management can only be accessed by the administrator. Therefore, the functionalities that we need to enhance are:

- **Restricting Deletion of Tales**: We have already implemented the functionality for deleting the published tales as a part of Tale management. However, the current implementation allows any user to delete the tales published by any other user. This is not desirable. Only the administrator should be able to delete the tales. Therefore, this functionality will restrict all other users except the administrator from deleting the tales.
- **Managing Comments**: The administrator may want to view, modify, and delete the comments that the users have submitted. This functionality will help the administrator to do so by providing complete access to all the functionalities of the Comment management module.

- **Searching for a User**: The administrator will need to look up a particular user with all his/her corresponding details, such as the tales published by him/her, comments given by him/her, and the profile of the user. The functionality to search for users will help the administrator to do so. The search will have an auto-complete feature so that the administrator needs to know only the first few characters of the user ID. Auto-completion is another feature that enhances the usability.

Now that we are clear with the requirements, let us move on to implementation.

## Implementing the Functionalities

Based on the requirements, the implementation can be divided into the following steps:

- Modification of the Deletion of Tales
- Providing access to all the functionalities of the Comment Management module
- Auto-completion of the User name during the User search
- Searching for a particular user

Let us start with the modification of the deletion of tales.

## Modification of the Deletion of Tales

We have to restrict the other users from deleting the published tales. To do so, we will show the link to delete the tale to the administrator only. That means, if the current user's role is not 'administrator', then the link will not be displayed. To achieve this end, we will have to make modifications in the `list.rhtml` file in `app/views/tales` folder. Open the file in your favorite editor. It will have the following code:

```
<h1>Listing tales</h1>
<table>
  <tr>
  <% for column in Tale.content_columns %>
    <th><%= h(column.human_name) %></th>
  <% end %>
  </tr>
<% for tale in @tales %>
  <tr>
  <% for column in Tale.content_columns %>
    <td><%=h tale.send(column.name) %></td>
  <% end %>
```

```
      <td><%= link_to 'Show', :action => 'show', :id => tale %></td>
      <td><%= link_to 'Edit', :action => 'edit', :id => tale %></td>
      <td><%= link_to 'Destroy', { :action => 'destroy', :id => tale },
      :confirm => 'Are you sure?', :method => :post %></td>
    </tr>
  <% end %>
  </table>
  <%= link_to 'Previous page', { :page => @tale_pages.current.previous }
  if @tale_pages.current.previous %>
  <%= link_to 'Next page', { :page => @tale_pages.current.next } if @
  tale_pages.current.next %>
  <br />
  <%= link_to 'New tale', :action => 'new' %>
```

Next, we will surround the `link_to 'Destroy'` with the `if` statement. In the `if` statement, we will check whether the role of the current user is `administrator` or not. If the role is `administrator`, we will display the link to delete the tales. We will get the role of the current user from the `session`. After the modifications, the code will be as follows:

```
  <h1>Listing tales</h1>
  <table>
    <tr>
    <% for column in Tale.content_columns %>
      <th><%= column.human_name %></th>
    <% end %>
    </tr>
  <% for tale in @tales %>
    <tr>
    <% for column in Tale.content_columns %>
      <td><%=h tale.send(column.name) %></td>
    <% end %>
      <td><%= link_to 'Show', :action => 'show', :id => tale %></td>
    <%if session[:role]=='administrator'%>
      <td><%= link_to 'Edit', :action => 'edit', :id => tale %></td>
      <td><%= link_to 'Destroy', { :action => 'destroy', :id => tale },
      :confirm => 'Are you sure?', :method => :post %></td>
      <%end%>
    </tr>
  <% end %>
  </table>
  <%= link_to 'Previous page', { :page => @tale_pages.current.previous }
  if @tale_pages.current.previous %>
  <%= link_to 'Next page', { :page => @tale_pages.current.next } if @
  tale_pages.current.next %>
  <br />
  %= link_to 'New tale', :action => 'new' %>
```

Next, let us move on to the comment functionality.

## Providing Access to All the Functionalities of the Comment Management Module

If you remember, we directly linked the 'adding new comment' functionality with the 'detailed view of tales', when we implemented the Comment management module. Now we will provide all the links to the administrator, including the list of comments. For that, we will add a link to the left menu so that the administrator can have access to all the functionalities of the Comment Management module. Open the `master.rhtml` file from the `app/views/layouts` folder. Add the following code just after the link for Genres:

```
<tr>
   <td><a
   href="http://localhost:3000/comment">Comments</a></td>
</tr>
```

After adding the link, the code of the left menu will be as follows:

```
<table width="75" border="1" cellpadding="2" cellspacing="2">
  <%if session[:role]=='administrator' %>
    <tr>
      <td width="286"><a href="http://localhost:3000/role">
      Roles</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/user/list">Users</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/genre">Genres</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/comment">Comments</a></td>
    </tr>
  <%end%>
    <tr>
      <td><a href="http://localhost:3000/tale">Tales</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/tag/tag_clouds">Browse By
      Tags</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/tag/search">Search By
      Tag</a></td>
    </tr>
    :
    :
</table>
```

With these changes, the second step for implementing the administrative interface is complete. Let us move on to the third step.

## Implementing Auto-Complete for the User name

TaleWiki will have many users. It is humanly impossible to remember each user's user name while trying to look up the information on a particular user. The auto-completion of user names will help the administrator to look up a particular name easily, by providing a list of user names starting with a given letter of the alphabet, and narrowing the list down to specific names, as the administrator types more letters of the name he or she is looking up.

To implement auto-complete, we will use the `text_field_with_auto_complete` helper. It accepts two arguments:

- **Name of the model**: The data will be retrieved and displayed as a list from the model whose name is passed as a value to this argument. In our case, it is the user.
- **Name of the field**: This accepts the attribute of the model (column of the table) whose data will be used to display the auto-completion list. In our case, it is the name field of the user table.

However, before implementing the auto-completion, let us add a link to the left menu so that the administrator can easily access the `Search User` functionality. After adding the link, the left menu will be as follows:

```
<table width="75" border="1" cellpadding="2" cellspacing="2">
  <%if session[:role]=='administrator' %>
    <tr>
      <td width="286"><a href="http://localhost:3000/role">
      Roles</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/user/list">Users</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/genre">Genres</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/comment">Comments</a></td>
    </tr>
    <tr>
      <td><a href="http://localhost:3000/user/search">Search
      User</a></td>
```

```
          </tr>
        <%end%>
        <tr>
          <td><a href="http://localhost:3000/tale">Tales</a></td>
        </tr>
        <tr>
          <td><a href="http://localhost:3000/tag/tag_clouds">Browse By
          Tags</a></td>
        </tr>
        <tr>
          <td><a href="http://localhost:3000/tag/search">Search By
          Tag</a></td>
        </tr>
        :
        :
    </table>
```

Next, open the `user_controller.rb` file from the `apps/controllers/` folder. Then add the following action method:

```
def search
end
```

Now create a file called `search.rhtml` in the `app/views/users` folder. Open it and write the following code:

```
<html>
<head>
<%=javascript_include_tag (:defaults) %>
</head>
<body>
<h1>Search User</h1>
<p>
<label for="user_name">Tag</label>
<%= text_field_with_auto_complete :user, :user_name  %> </p>
<br/>
</body>
</html>
```

Next, open the `user_controller.rb`. Add the following statement after the `in_place_editor_for` helper:

```
auto_complete_for :user, :user_name
```

This is the helper at the controller level, corresponding to the auto-completion helper on the view side. It retrieves data from the model on the basis of passed arguments, and sends the data back to the view. In our case, the model is `user` and the attribute is `user_name`. The `auto_complete_for` helper takes the user name from the user table and passes the data back to the search page. That completes the auto-complete feature. Next, let us move on to the implementation of the search functionality.

## Implementing Search

The search functionality will not only provide details of the user, but also the details of the stories published by the user as well as comments given by him or her. The implementation that we will do now will display the user details and the tales published by him/her. The second part, that is displaying the comments of a user, is left as an exercise for you. So, let us begin the implementation.

As a first step in implementing the search, we will modify the `search.rhtml` file. We will have to send the selected user name to the controller. To do so, we will have to place the textbox in a form. So, open the `search.rhtml` file and surround the textbox with the form helper. So that it looks as follows:

```
<html>
<head>
<%=javascript_include_tag (:defaults) %>
</head>
<body>
<h1>Search User</h1>
<p>
<% form_tag :action=>'search_result'%>
<label for="user_name">Tag</label>
<%= text_field_with_auto_complete :user, :user_name  %> </p>
<% submit_tag "Search" %>
<% end %>
<br/>
</body>
</html>
```

The form tag is calling an action method named `search_result`. Therefore, next we will implement the `search_result` method. In the `user_controller`, add the following method:

```
def search_result
     user_to_find=params[:user]
     user1=user_to_find[:user_name]
     @user=User.find(:first, :conditions => ["user_name = ?",
     user1])
     @tales=@user.tale
end
```

As, we are not passing the ID of the user in the search page, the parameters are passed as part of a hash. So, first we retrieve the hash containing the user details. Following the principle of Convention over Configuration, the key of the hash is 'user'. The value of this key is another hash containing the parameter that we require. The key in the second hash is the parameter name—`user_name`—which is used to retrieve the value entered by the administrator. After that, using the `find` method, the user details are retrieved. This is similar to what we did while assigning the user.

Next, let us design the search result page. The search result page will have two sections. The first section will have the details of the user, and the second section will have the details of the tales published by the user. As the details of the user can be accessed from the user object, the first section will be as follows:

```
<table width="100%" border="1">
  <tr>
    <th scope="row"> User Name </th>
    <td width="71"><%=@user.user_name%></td>
  </tr>
  <tr>
    <th scope="row">Password</th>
    <td><%=@user.password%></td>
  </tr>
  <tr>
    <th scope="row">First Name </th>
    <td><%=@user.first_name%></td>
  </tr>
  <tr>
    <th scope="row">Last Name </th>
    <td><%=@user.last_name%></td>
  </tr>
  <tr>
    <th scope="row">Age</th>
    <td><%=@user.age%></td>
  </tr>
  <tr>
    <th scope="row">eMail</th>
    <td><%=@user.email%></td>
  </tr>
  <tr>
    <th scope="row">Country</th>
    <td><%=@user.country%></td>
  </tr>
  <tr>
    <th scope="row">&nbsp;</th>
    <td>&nbsp;</td>
  </tr>
</table>
```

Create a new file named `search_result.rhtml` in the `app/views/user` folder. Open it and add the code just given. Next, let us create the second section using the `tale` object. Place the following code after the `<table>` tag of the `search_result.rhtml`:

```
<table width="100%">
  <tr>
    <td><b> Tale Details</b></td>
  </tr>
</table>
<table width="100%" border="1">
  <tr>
    <th width="11%" scope="col" align="left">Title</th>
    <th width="10%" scope="col" align="left">Tale</th>
    <th width="30%" scope="col" align="left">Submission Date </th>
    <th width="21%" scope="col" align="left">Source</th>
    <th width="14%" scope="col" align="left">Status</th>
    <th width="14%" scope="col" align="left">Genre</th>
  </tr>
  <% @tales.each do |tale| %>
  <tr>
    <td><%=tale.title%></td>
    <td><%=tale.body_text%></td>
    <td><%=tale.submission_date%></td>
    <td><%=tale.source%></td>
    <td><%=tale.status%></td>
    <td><%=tale.genre.name%></td>
  </tr>
  <%end%>
</table>
```

It iterates over the array of tales and displays the details of the individual tales. That completes the search functionality. Next, let us test our modifications.

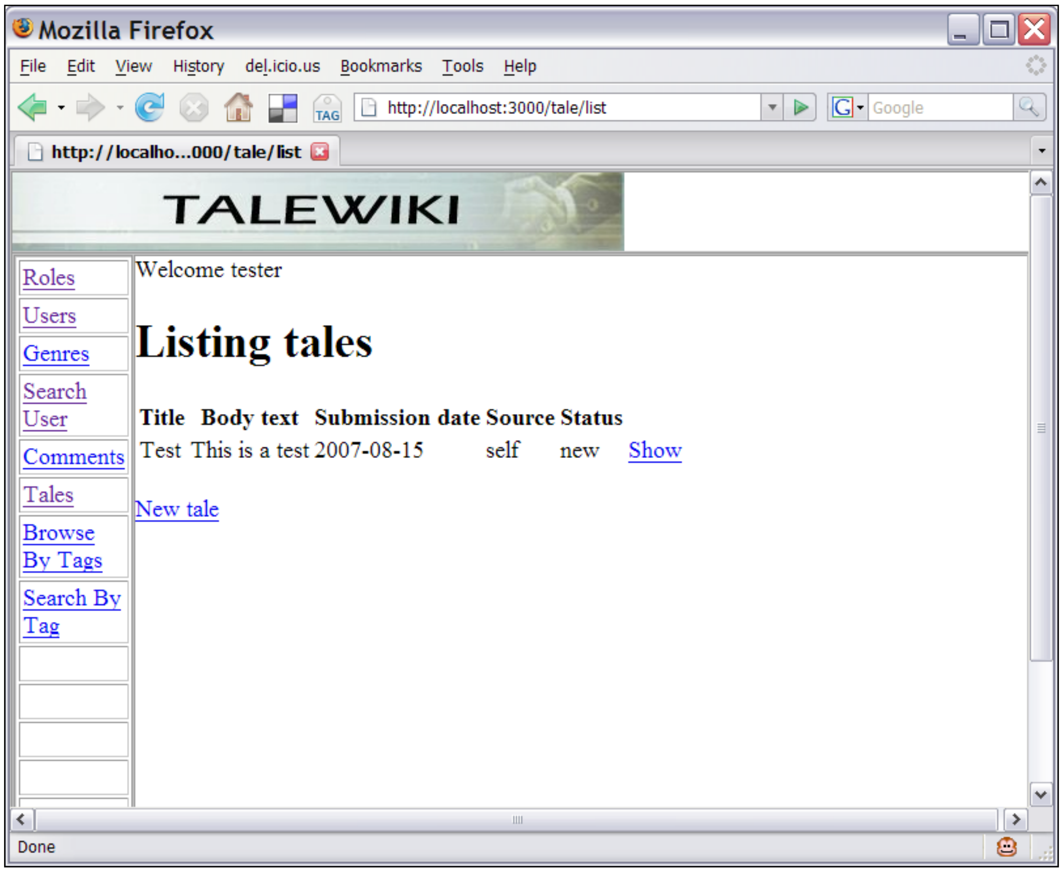
## Testing the Modifications

First, let us test the modification to the list functionality of the Tale Management module. Log in with the following user name and password:

**Username**: **tester**

**Password**: **testing123**

Click on the **Tales** link. If you get the following screen, then the modifications are working well:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/tale/list
Google
http://localho...000/tale/list
TALEWIKI
Roles
Users
Genres
Search User
Comments
Tales
Browse By Tags
Search By Tag
Done

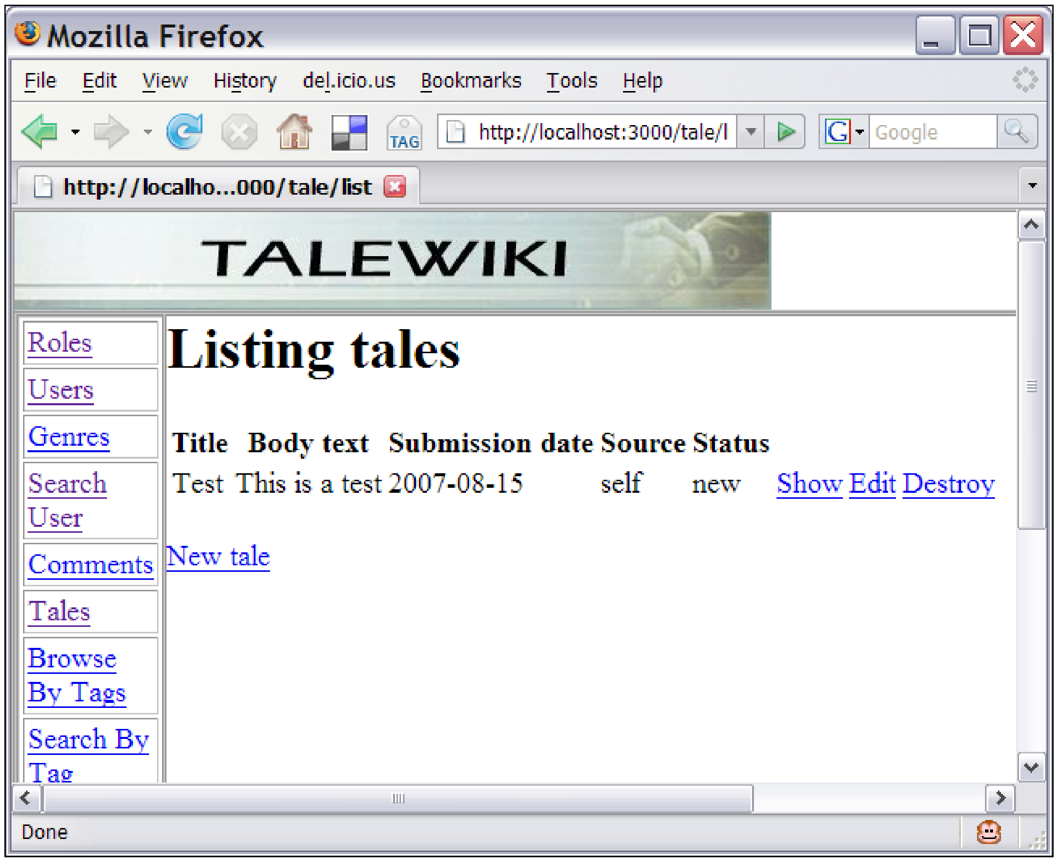
Welcome tester

Listing tales

| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show |

New tale

Next, log in with the administrator username and password. If you get the following screen, then the administrative interface is working fine:

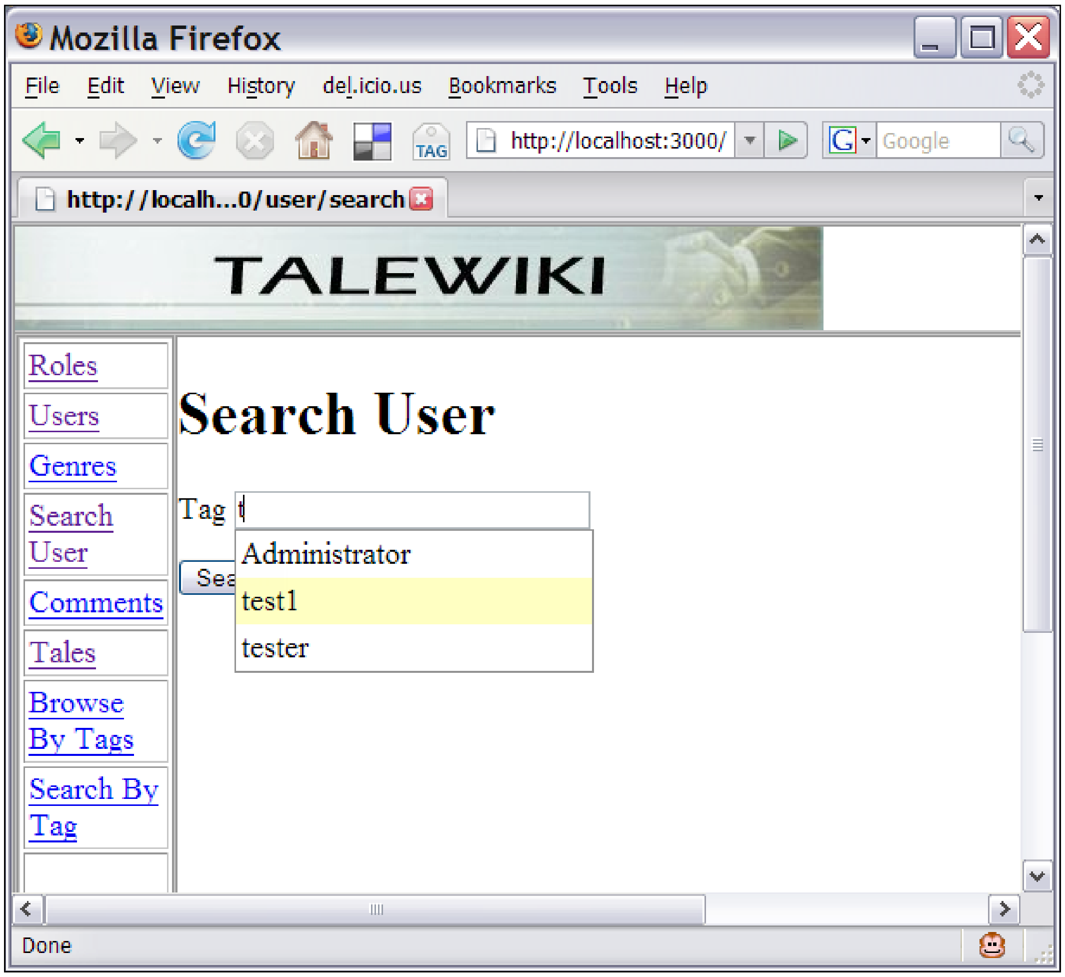

Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/tale/l
Google
http://localho...000/tale/list
TALEWIKI
Roles
Users
Genres
Search User
Comments
Tales
Browse By Tags
Search By Tag
Done

Listing tales

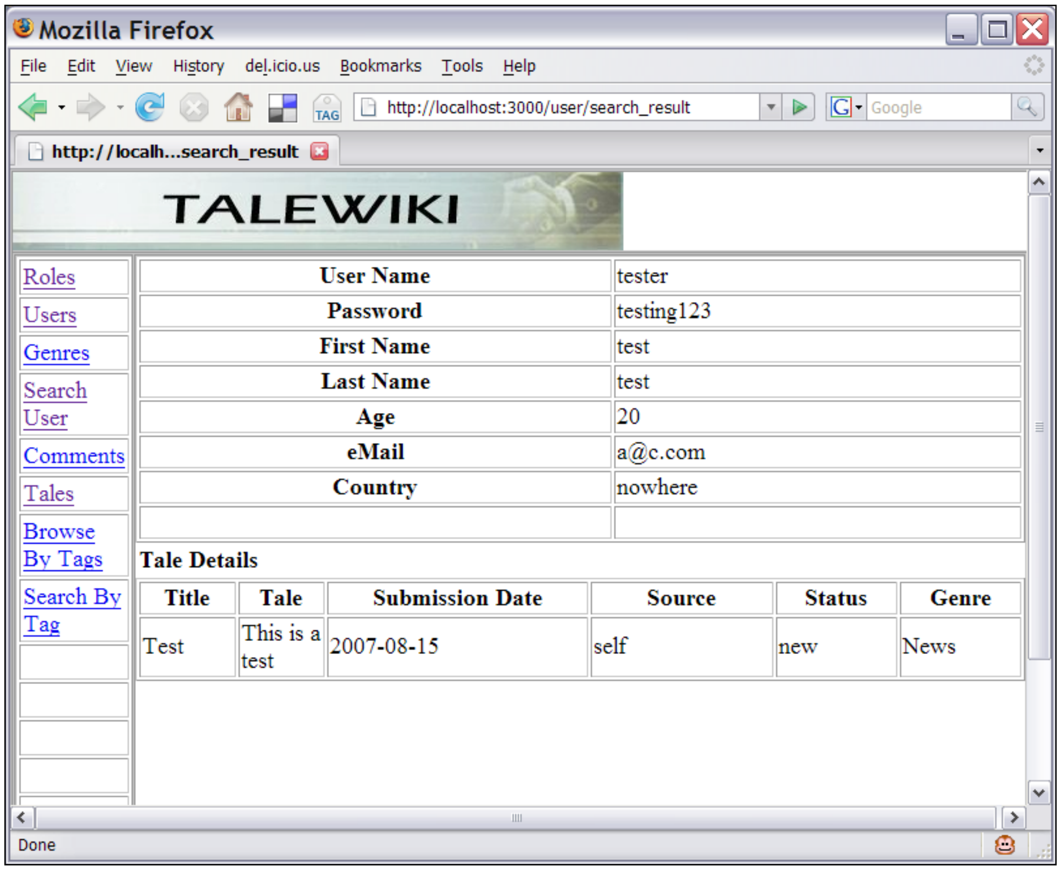
| Title | Body text | Submission date | Source | Status | |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | Show Edit Destroy |

New tale

Next, click on the **Search user** link. On the next page, input **t** in the text field. If you get the following screen, then the auto-completion is working fine:



Select **tester** and press enter. If you get the following screen, then the search is functioning well:


Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost:3000/user/search_result
Google
http://localh...search_result
TALEWIKI
Roles
Users
Genres
Search User
Comments
Tales
Browse By Tags
Search By Tag
Done

| | |
|---|---|
| **User Name** | tester |
| **Password** | testing123 |
| **First Name** | test |
| **Last Name** | test |
| **Age** | 20 |
| **eMail** | a@c.com |
| **Country** | nowhere |
| | |

**Tale Details**

| Title | Tale | Submission Date | Source | Status | Genre |
|---|---|---|---|---|---|
| Test | This is a test | 2007-08-15 | self | new | News |

With that, we come to the end of this chapter.

## Summary

This chapter marks the last step in the development of TaleWiki. In the next chapter, we will look at the steps in deploying our website. However, if you compare the development that we have done with the real-world scenario, it is the first phase or the 'basic setup' phase. The next level is to add the services that will make the website unique. Some of the services that can add value to any website are:

- **Polling**: Understanding the mood of the user is essential for any social networking site. Polls are one of the ways to gauge the user's opinion.
- **Personalization**: Each user is an individual. Hence, their tastes with respect to the themes, color schemes, etc., may differ. Personalization can help you to attract more users.

You can implement these services to test the understanding of the concepts that we have discussed until now. With that, I will wrap up this chapter. Keep reading to understand the deployment procedure!

# 10
# Deploying the TaleWiki

Once the development and testing is completed, the next step (which is also the last step before going live), is to deploy the application. In this chapter, we will see how to deploy the TaleWiki. In the first section, we will discuss the difference between the development mode and the deployment/production mode. In the second section, we will configure Mongrel so that it can run in the production mode. We will wrap up with the points to keep in mind while running a RoR-based website.

## Understanding the Production Environment

Before going into the details about the production environment, let us have a look at what an 'Environment' really means. The environment that we are discussing here is with respect to the Mongrel server. The Mongrel server can run in three different modes:

- Development Mode
- Test Mode
- Production Mode

The mode in which the Mongrel server runs for a RoR site is known as the environment. In other words, mode is for the server and environment is for the site. In our case, the Mongrel server is running in the development mode and the TaleWiki is running in the development environment. Keeping this in mind, let us look at the different modes in which the Mongrel can run.

## Development Mode

The development mode is the default mode for the Mongrel server. In this mode, Mongrel provides the development environment for the application or the site. In this environment, ease-of-use is given importance over speed or scalability. Hence, for every request, the application is reloaded by the server. This is the reason that you don't need to restart the server when you make changes to the application. In addition, caching is disabled when an application is using the development environment. This essentially means that none of the caching services is available in this environment.

The other important aspect of the development environment is the information it provides when application fails to service a request. In this environment, whenever a request fails, Rails provide you all the information related to the failed request. The information is displayed on the browser.

## Test Mode

Test mode provides the test environment for an application. It is just like the development environment with one difference—the database is recreated every time test cases are run. Therefore, whenever you run a test using the testing service, then it is necessary to create a different database with the same set of tables.

The second important point you have to keep in mind while using the test environment is regarding services, such as mailing. For example, in the test environment, mails are not delivered to the mail server. The mail delivery is just simulated. We haven't used the testing service in this book. All the testing that we have done so far is manual.

## Production Mode

When the Mongrel server is run in the production mode, it provides the production environment for your application. In the production environment, speed and scalability is given importance over ease-of-use. Therefore, the application is not reloaded for each request. In this mode, Mongrel ensures that once all the classes related to the Model and Controller are loaded, they are not reloaded. Secondly, caching is enabled for any application using the production environment. In other words, your application can make use of the caching services provided by RoR.

In addition, the debugging information is also not provided whenever any failure occurs. Instead of details of the failed request, Mongrel shows the `500.rhtml` page in the `public` folder. This ensures that the details of the application are not given out to the public, if a request fails. The `500` is related to the HTTP protocol that tells the user that the service is currently unavailable. The `500.rhtml` displays the 'service unavailable' message in the case of a request failure.

Now that you have understood the differences between the three modes, let us look at how to move from the development environment to the production environment.

## Changing to the Production Environment

To change from the development environment to the production environment, you will need to do the following things:

- Migrating to the Production database
- Configuring Mongrel to start in the production mode

For the former, migration files can be very helpful.

## Migrating to the Production Database

To migrate to the Production database, you will need to create a database named `talewiki_production`. Then, only RoR will use the production database. This is due to the 'Convention-over-Configuration' philosophy of RoR. When Mongrel runs in the production mode, it looks for the database whose name ends with `_production`. It will work only when such a database is found.

In order to migrate to the production database, first create a database named `talewiki_production`. Then open the `database.yml` file from the `config` folder. Modify it so that the content looks as follows:

```
#MySQL (default setup).  Versions 4.1 and 5.0 are recommended.
#
#Install the MySQL driver:
#   gem install mysql
#On MacOS X:
#   gem install mysql -- --include=/usr/local/lib
#On Windows:
#   gem install mysql
#       Choose the win32 build.
#       Install MySQL and put its /bin directory on your path.
#
#And be sure to use new-style password hashing:
```

```
#   http://dev.mysql.com/doc/refman/5.0/en/old-client.html
production:
   adapter: mysql
   database: talewiki_production
   username: root
   password:
   host: localhost
```

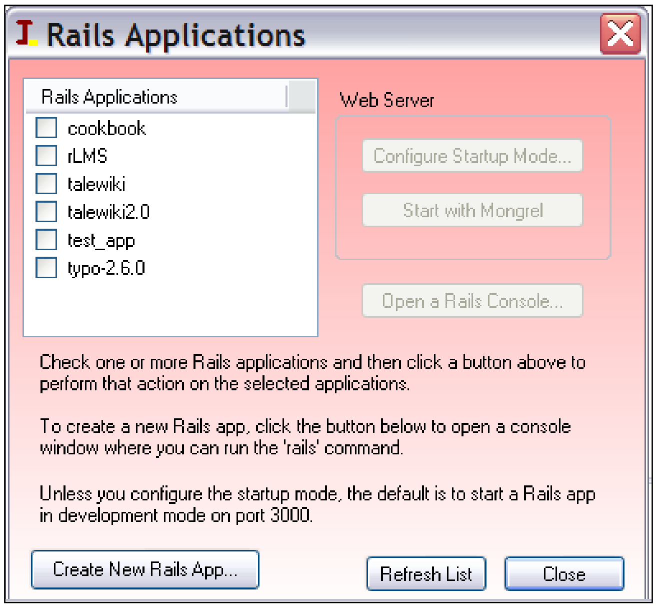
We have removed the development and test entries. This is for security purposes, as having the development database entry along with the production database entry can help a hacker to know the database schema. Next, at the command prompt give the following command:

```
C:\InstantRails\rails_app\talewiki>rake db:migrate
```

The `migrate` task will now create all the tables in the production database. This completes the migration of the database.

## Configuring Mongrel

To use the speed and caching services, you need to tell RoR to run in the production mode. In order to do it, you will have to set the value of the `RAILS_ENV` variable to the string `production`. Instant Rails provides a very easy tool to do exactly this. Select **Rails Application** | **Manage Rails Application...** from the Instant Rails menu. You will get the following dialog box:

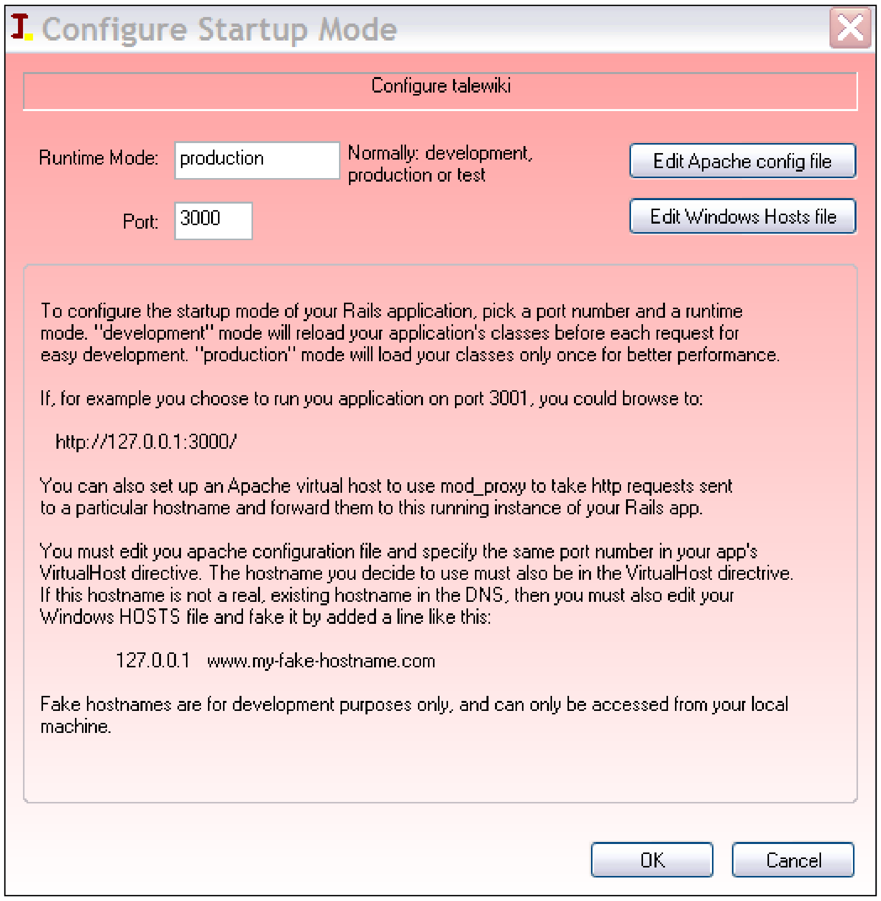


Check one or more Rails applications and then click a button above to perform that action on the selected applications.

To create a new Rails app, click the button below to open a console window where you can run the 'rails' command.

Unless you configure the startup mode, the default is to start a Rails app in development mode on port 3000.

Select the `talewiki` checkbox and click on the `Configure Startup Mode...` button. You will see the following window:



Runtime Mode: production

Normally: development, production or test

Port: 3000

To configure the startup mode of your Rails application, pick a port number and a runtime mode. "development" mode will reload your application's classes before each request for easy development. "production" mode will load your classes only once for better performance.

If, for example you choose to run you application on port 3001, you could browse to:

http://127.0.0.1:3000/

You can also set up an Apache virtual host to use mod_proxy to take http requests sent to a particular hostname and forward them to this running instance of your Rails app.

You must edit you apache configuration file and specify the same port number in your app's VirtualHost directive. The hostname you decide to use must also be in the VirtualHost directive. If this hostname is not a real, existing hostname in the DNS, then you must also edit your Windows HOSTS file and fake it by added a line like this:

```
127.0.0.1    www.my-fake-hostname.com
```

Fake hostnames are for development purposes only, and can only be accessed from your local machine.

In the textbox labelled **Runtime Mode**, enter the value as **production** and click **OK**. The Mongrel server will now run in the production mode. That completes the configuring of Mongrel.

To change back to the development or test mode, give either **development** or **test** in the textbox.

## Points to Consider

There are several points to keep in mind after deploying an application. Two of the most important points are:

- **Scaling**: When your site's traffic is limited to about 1000 users (or 1000 hits), Mongrel performs well. The performance of a server with respect to the number of user requests is known as scaling. However, if it crosses the limit of 1000 users, then Mongrel does not scale well. In that case, you will have to run Mongrel in conjunction with other web-servers, such as Apache with CGI, Apache with FastCGI, and lighttpd with FastCGI. Among these, Apache with FastCGI is an industry standard, as Apache is the most robust of them all. However, configuring Apache is a complex task. Therefore, if you need to scale up, lighttpd with FastCGI is a good option.
- **Bottlenecks**: As scaling is to the web or application server, so are bottlenecks to the database servers. When the application's response becomes sluggish due to the time taken in querying the database, then the application is said to have hit a bottleneck. To overcome bottlenecks, two basic strategies can be applied. The first of them is to bring most of the database processing to the application level. Instead of relying on the database to do the complex multi-table based computations, the application itself is upgraded to handle the computation. In other words, the database is simply used to store and retrieve data. The manipulation of data is completely taken care of by the logic embedded in the application itself. The second strategy is to use the native SQL. RoR's ORM library is pretty optimized. However, there are situations where using native SQL can be more effective than the generic optimization provided by Active Record. You will have to decide which strategy to apply depending on the situation.

These are two main points that you will have to keep track of. However, keep in mind that these situations will not arise unless you deploy the application.

## Summary

With that, we come to the end of the chapter. Throughout the book, we have seen how easily one can develop dynamic sites using RoR, without sacrificing the flexibility and robustness. However, this is just an overview of what RoR can do. Based on what we have discussed in this book, you can create your own commercial sites. The only point you have to keep in mind is that RoR is an evolving framework. Therefore, every other day some new functionality is being added. Keep track of that and you can create highly scalable and robust sites without sidetracking usability and flexibility. That is the beauty of RoR and that is what this book has tried to convey. With that I conclude this chapter as well as this book.

# Index