HHLinuxClub

Ruby on Rails Course 2009

Getting Started with Rails

Original authors:
James Miller
Mike Gunderloy
Dave Rothlisberger

Edited by:
João Cardoso
Matias Korhonen

Source: http://guides.rubyonrails.org/

Table of Contents

Getting Started with Rails

This guide covers getting up and running with Ruby on Rails. After reading it, you should be familiar with:

- Installing Rails, creating a new Rails application, and connecting your application to a database
- The general layout of a Rails application
- The basic principles of MVC (Model, View Controller) and RESTful design
- How to quickly generate the starting pieces of a Rails application.

This Guide is based on Rails 2.3. Some of the code shown here will not work in older versions of Rails.

This Guide Assumes

This guide is designed for beginners who want to get started with a Rails application from scratch. It does not assume that you have any prior experience with Rails. However, to get the most out of it, you need to have some prerequisites installed:

- The Ruby language
- The RubyGems packaging system
- A working installation of SQLite (preferred), MySQL, or PostgreSQL

It is highly recommended that you **familiarize yourself with Ruby before diving into Rails**. You will find it much easier to follow what's going on with a Rails application if you understand basic Ruby syntax. Rails isn't going to magically revolutionize the way you write web applications if you have no experience with the language it uses.

What is Rails?

Rails is a web development framework written in the Ruby language. It is designed to make programming web applications easier by making several assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks. Longtime Rails developers also report that it makes web application development more fun.

Rails is opinionated software. That is, it assumes that there is a best way to do things, and it's designed to encourage that best way – and in some cases to discourage alternatives. If you learn "The Rails Way" you'll probably discover a tremendous increase in productivity. If you persist in bringing old habits from other languages to your Rails development, and trying to use patterns you learned elsewhere, you may have a less happy experience.

The Rails philosophy includes several guiding principles:

- DRY – "Don't Repeat Yourself" – suggests that writing the same code over and over again is a bad thing.
- Convention Over Configuration – means that Rails makes assumptions about what you want to do and how you're going to do it, rather than letting you tweak every little thing through endless configuration files.
- REST is the best pattern for web applications – organizing your application around resources and standard HTTP verbs is the fastest way to go.

The MVC Architecture

Rails is organized around the Model, View, Controller architecture, usually just called MVC. MVC benefits include:

- Isolation of business logic from the user interface
- Ease of keeping code DRY
- Making it clear where different types of code belong for easier maintenance

Models

A model represents the information (data) of the application and the rules to manipulate that data. In the case of Rails, models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, one table in your database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models.

Views

Views represent the user interface of your application. In Rails, views are often HTML files with embedded Ruby code that performs tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from your application.

Controllers

Controllers provide the "glue" between models and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.

The Components of Rails

Rails provides a full stack of components for creating web applications, including:

- Action Controller
- Action View
- Active Record
- Action Mailer
- Active Resource
- Railties
- Active Support

Action Controller

Action Controller is the component that manages the controllers in a Rails application. The Action Controller framework processes incoming requests to a Rails application, extracts parameters, and dispatches them to the intended action. Services provided by Action Controller include session management, template rendering, and redirect management.

Action View

Action View manages the views of your Rails application. It can create both HTML and XML output by default. Action View manages rendering templates, including nested and partial templates, and includes built-in AJAX support.

Active Record

Active Record is the base for the models in a Rails application. It provides database independence, basic CRUD functionality, advanced finding capabilities, and the ability to relate models to one another, among other services.

Action Mailer

Action Mailer is a framework for building e-mail services. You can use Action Mailer to send emails based on flexible templates, or to receive and process incoming email.

Active Resource

Active Resource provides a framework for managing the connection between business objects an RESTful web services. It implements a way to map web-based resources to local objects with CRUD semantics.

Railties

Railties is the core Rails code that builds new Rails applications and glues the various frameworks together in any Rails application.

Active Support

Active Support is an extensive collection of utility classes and standard Ruby library extensions that are used in the Rails, both by the core code and by your applications.

REST

REST, an acronym for Representational State Transfer, boils down to two main principles for our purposes:

- Using resource identifiers (which, for the purposes of discussion, you can think of as URLs) to represent resources
- Transferring representations of the state of that resource between system components.

For example, to a Rails application a request such as this:

```
DELETE /photos/17
```

would be understood to refer to a photo resource with the ID of 17, and to indicate a desired action – deleting that resource. REST is a natural style for the architecture of web applications, and Rails makes it even more natural by using conventions to shield you from some of the RESTful complexities and browser quirks.

Creating a New Rails Project

If you follow this guide, you'll create a Rails project called `blog`, a (very) simple weblog. Before you can start building the application, you need to make sure that you have Rails itself installed.

Installing Rails

In most cases, the easiest way to install Rails is to take advantage of RubyGems:

```
$ gem install rails
```

There are some special circumstances in which you might want to use an alternate installation strategy:

- If you're working on Windows, you may find it easier to install Instant Rails. Be aware, though, that Instant Rails releases tend to lag seriously behind the actual Rails version. Also, you will find that Rails development on Windows is overall less pleasant than on other operating systems. If at all possible, we suggest that you install a Linux virtual machine and use that for Rails development, instead of using Windows.
- If you want to keep up with cutting-edge changes to Rails, you'll want to clone the Rails source code from GitHub. This is not recommended as an option for beginners, though.

Creating the Blog Application

Open a terminal, navigate to a folder where you have rights to create files, and type:

```
$ rails blog
```

This will create a Rails application that uses a SQLite database for data storage. If you prefer to use MySQL, run this command instead:

```
$ rails blog -d mysql
```

And if you're using PostgreSQL for data storage, run this command:

```
$ rails blog -d postgresql
```

TIP: You can see all of the switches that the Rails application builder accepts by running `rails -h`.

After you create the blog application, switch to its folder to continue work directly in that application:

```
$ cd blog
```

In any case, Rails will create a folder in your working directory called `blog`. Open up that folder and explore its contents. Most of the work in this tutorial will happen in the `app/` folder, but here's a basic rundown on the function of each folder that Rails creates in a new application by default:

File/Folder	**Purpose**
`README`	This is a brief instruction manual for your application. Use it to tell others what your application does, how to set it up, and so on.
`Rakefile`	This file contains batch jobs that can be run from the terminal.
`app/`	Contains the controllers, models, and views for your application. You'll focus on this folder for the remainder of this guide.
`config/`	Configure your application's runtime rules, routes, database, and more.
`db/`	Shows your current database schema, as well as the database migrations. You'll learn about migrations shortly.
`doc/`	In-depth documentation for your application.
`lib/`	Extended modules for your application (not covered in this guide).
`log/`	Application log files.
`public/`	The only folder seen to the world as-is. This is where your images, javascript, stylesheets (CSS), and other static files go.
`script/`	Scripts provided by Rails to do recurring tasks, such as benchmarking, plugin installation, and starting the console or the web server.
`test/`	Unit tests, fixtures, and other test apparatus.
`tmp/`	Temporary files
`vendor/`	A place for third-party code. In a typical Rails application, this includes Ruby Gems, the Rails source code (if you install it into your project) and plugins containing additional prepackaged functionality.

Configuring a Database

Just about every Rails application will interact with a database. The database to use is specified in a configuration file, `config/database.yml`. If you open this file in a new Rails application, you'll see a default database configuration using SQLite. The file contains sections for three different environments in which Rails can run by default:

- The `development` environment is used on your development computer as you interact manually with the application
- The `test` environment is used to run automated tests
- The `production` environment is used when you deploy your application for the world to use.

Configuring a SQLite Database

Rails comes with built-in support for SQLite, which is a lightweight serverless database application. While a busy production environment may overload SQLite, it works well for development and testing. Rails defaults to using a SQLite database when creating a new project, but you can always change it later.

Here's the section of the default configuration file with connection information for the development environment:

```
development:
        adapter: sqlite3
        database: db/development.sqlite3
        pool: 5
        timeout: 5000
```

If you don't have any database set up, SQLite is the easiest to get installed. If you're on OS X 10.5 or greater on a Mac, you already have it. Otherwise, you can install it using RubyGems:

```
$ gem install sqlite3-ruby
```

Configuring a MySQL Database

If you choose to use MySQL, your `config/database.yml` will look a little different. Here's the development section:

```
development:
        adapter: mysql
        encoding: utf8
        database: blog_development
        pool: 5
        username: root
        password:
        socket: /tmp/mysql.sock
```

If your development computer's MySQL installation includes a root user with an empty password, this configuration should work for you. Otherwise, change the username and password in the development section as appropriate.

Configuring a PostgreSQL Database

If you choose to use PostgreSQL, your `config/database.yml` will be customized to use PostgreSQL databases:

```
development:
        adapter: postgresql
        encoding: unicode
        database: blog_development
        pool: 5
        username: blog
        password:
```

Change the username and password in the `development` section as appropriate.

Creating the Database

Now that you have your database configured, it's time to have Rails create an empty database for you. You can do this by running a rake command:

```
$ rake db:create
```

Rake is a general-purpose command-runner that Rails uses for many things. You can see the list of available rake commands in your application by running `rake -T`.

Hello, Rails!

One of the traditional places to start with a new language is by getting some text up on screen quickly. To do that in Rails, you need to create at minimum a controller and a view. Fortunately, you can do that in a single command. Enter this command in your terminal:

```
$ script/generate controller home index
```

TIP: If you're on Windows, or your Ruby is set up in some non-standard fashion, you may need to explicitly pass Rails `script` commands to Ruby: `ruby script/generate controller home index`.

Rails will create several files for you, including `app/views/home/index.html.erb`. This is the template that will be used to display the results of the `index` action (method) in the home controller. Open this file in your text editor and edit it to contain a single line of code:

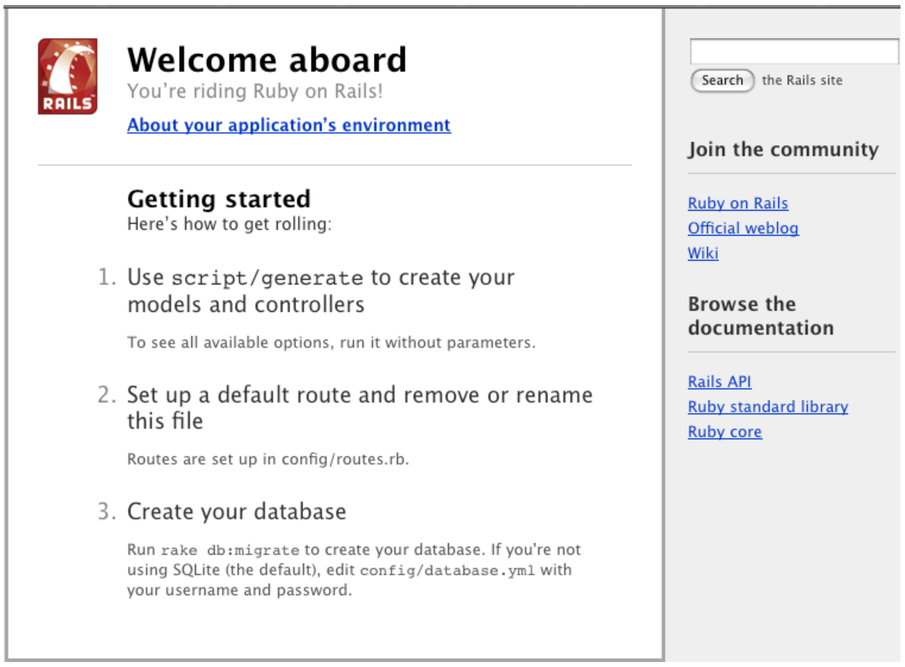
```
<h1>Hello, Rails!</h1>
```

Starting up the Web Server

You actually have a functional Rails application already – after running only two commands! To see it, you need to start a web server on your development machine. You can do this by running another command:

```
$ script/server
```

This will fire up an instance of the Mongrel web server by default (Rails can also use several other web servers). To see your application in action, open a browser window and navigate to `http://localhost:3000`. You should see Rails' default information page:



Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Getting started

Here's how to get rolling:

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

   To see all available options, run it without parameters.

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

   Routes are set up in config/routes.rb.

3. Create your database

   Run `rake db:migrate` to create your database. If you're not using SQLite (the default), edit `config/database.yml` with your username and password.

TIP: To stop the web server, hit Ctrl+C in the terminal window where it's running. In development mode, Rails does not generally require you to stop the server; changes you make in files will be automatically picked up by the server.

The "Welcome Aboard" page is the *smoke test* for a new Rails application: it makes sure that you have your software configured correctly enough to serve a page. To view the page you just created, navigate to `http://localhost:3000/home/index`.

Setting the Application Home Page

You'd probably like to replace the "Welcome Aboard" page with your own application's home page. The first step to doing this is to delete the default page from your application:

```
$ rm public/index.html
```

Now, you have to tell Rails where your actual home page is located. Open the file `config/routes.rb` in your editor. This is your application's, *routing file*, which holds entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions. At the bottom of the file you'll see the *default routes*:

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

The default routes handle simple requests such as `/home/index`: Rails translates that into a call to the `index` action in the home controller. As another example, `/posts/edit/1` would run the `edit` action in the posts controller with an id of 1.

To hook up your home page, you need to add another line to the routing file, above the default routes:

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

This line illustrates one tiny bit of the "convention over configuration" approach: if you don't specify an action, Rails assumes the `index` action.

Now if you navigate to `http://localhost:3000` in your browser, you'll see the `home/index` view.

Getting Up and Running Quickly With Scaffolding

Rails *scaffolding* is a quick way to generate some of the major pieces of an application. If you want to create the models, views, and controllers for a new resource in a single operation, scaffolding is the tool for the job.

Creating a Resource

In the case of the blog application, you can start by generating a scaffolded Post resource: this will represent a single blog posting. To do this, enter this command in your terminal:

```
$ script/generate scaffold Post name:string title:string content:text
```

While scaffolding will get you up and running quickly, the "one size fits all" code that it generates is unlikely to be a perfect fit for your application. In most cases, you'll need to customize the generated code. Many experienced Rails developers avoid scaffolding entirely, preferring to write all or most of their source code from scratch.

The scaffold generator will build 14 files in your application, along with some folders, and edit one more. Here's a quick overview of what it creates:

File	Purpose
`app/models/post.rb`	The Post model
`db/migrate/20090113124235_create_posts.rb`	Migration to create the posts table in your database (your name will include a different timestamp)
`app/views/posts/index.html.erb`	A view to display an index of all posts
`app/views/posts/show.html.erb`	A view to display a single post
`app/views/posts/new.html.erb`	A view to create a new post
`app/views/posts/edit.html.erb`	A view to edit an existing post
`app/views/layouts/posts.html.erb`	A view to control the overall look and feel of the other posts views
`public/stylesheets/scaffold.css`	Cascading style sheet to make the scaffolded views look better
`app/controllers/posts_controller.rb`	The Posts controller
`test/functional/posts_controller_test.rb`	Functional testing harness for the posts controller
`app/helpers/posts_helper.rb`	Helper functions to be used from the posts views
`config/routes.rb`	Edited to include routing information for posts
`test/fixtures/posts.yml`	Dummy posts for use in testing
`test/unit/post_test.rb`	Unit testing harness for the posts model
`test/unit/helpers/posts_helper_test.rb`	Unit testing harness for the posts helper

Running a Migration

One of the products of the `script/generate scaffold` command is a *database migration*. Migrations are Ruby classes that are designed to make it simple to create and modify database tables. Rails uses rake commands to run migrations, and it's possible to undo a migration after it's been applied to your database. Migration filenames include a timestamp to ensure that they're processed in the order that they were created.

If you look in the db/migrate/20090113124235_create_posts.rb file (remember, yours will have a slightly different name), here's what you'll find:

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.string :name
      t.string :title
      t.text :content

      t.timestamps
    end
  end

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

If you were to translate that into words, it says something like: when this migration is run, create a table named posts with two string columns (name and title) and a text column (content), and generate timestamp fields to track record creation and updating.

At this point, you can use a rake command to run the migration:

```
$ rake db:migrate
```

Remember, you can't run migrations before running rake db:create to create your database, as we covered earlier.

Because you're working in the development environment by default, this command will apply to the database defined in the development section of your config/database.yml file.

Adding a Link

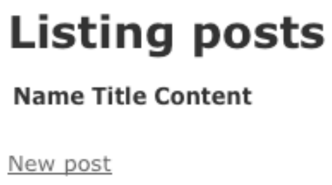
To hook the posts up to the home page you've already created, you can add a link to the home page. Open /app/views/home/index.html.erb and modify it as follows:

```
<h1>Hello, Rails!</h1>
<%= link_to "My Blog", posts_path %>
```

The link_to method is one of Rails' built-in view helpers. It creates a hyperlink based on text to display and where to go – in this case, to the path for posts.

Working with Posts in the Browser

Now you're ready to start working with posts. To do that, navigate to http://localhost:3000 and then click the "My Blog" link:



Listing posts

Name Title Content

New post

This is the result of Rails rendering the `index` view of your posts. There aren't currently any posts in the database, but if you click the `New Post` link you can create one. After that, you'll find that you can edit posts, look at their details, or destroy them. All of the logic and HTML to handle this was built by the single `script/generate scaffold` command.

TIP: In development mode (which is what you're working in by default), Rails reloads your application with every browser request, so there's no need to stop and restart the web server.

Congratulations, you're riding the rails! Now it's time to see how it all works.

The Model

The model file, `app/models/post.rb` is about as simple as it can get:

```
class Post < ActiveRecord::Base
end
```

There isn't much to this file – but note that the `Post` class inherits from `ActiveRecord::Base`. Active Record supplies a great deal of functionality to your Rails models for free, including basic database CRUD (Create, Read, Update, Destroy) operations, data validation, as well as sophisticated search support and the ability to relate multiple models to one another.

Adding Some Validation

Rails includes methods to help you validate the data that you send to models. Open the `app/models/post.rb` file and edit it:

```
class Post < ActiveRecord::Base
  validates_presence_of :name, :title
  validates_length_of :title, :minimum => 5
end
```

These changes will ensure that all posts have a name and a title, and that the title is at least five characters long. Rails can validate a variety of conditions in a model, including the presence or uniqueness of columns, their format, and the existence of associated objects.

Using the Console

To see your validations in action, you can use the console. The console is a command-line tool that lets you execute Ruby code in the context of your application:q

```
$ script/console
```

After the console loads, you can use it to work with your application's models:

```
>> p = Post.create(:content => "A new post")
=> #<Post id: nil, name: nil, title: nil, content: "A new post",
created_at: nil, updated_at: nil>

>> p.save
=> false

>> p.errors
=> #<ActiveRecord::Errors:0x23bcf0c @base=#<Post id: nil, name: nil, title:
nil, content: "A new post", created_at: nil, updated_at: nil>, @errors=
{"name"=>["can't be blank"], "title"=>["can't be blank", "is too short
(minimum is 5 characters)"]}>
```

This code shows creating a new `Post` instance, attempting to save it and getting `false` for a return value (indicating that the save failed), and inspecting the `errors` of the post.

Unlike the development web server, the console does not automatically load your code afresh for each line. If you make changes to your models while the console is open, type `reload!` at the console prompt to load them.

Listing All Posts

The easiest place to start looking at functionality is with the code that lists all posts. Open the file `app/controllers/posts_controller.rb` and look at the `index` action:

```
def index
  @posts = Post.find(:all)

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

This code sets the `@posts` instance variable to an array of all posts in the database. `Post.find(:all)` or `Post.all` calls the `Post` model to return all of the posts that are currently in the database, with no limiting conditions.

The `respond_to` block handles both HTML and XML calls to this action. If you browse to http://localhost:3000/posts.xml, you'll see all of the posts in XML format. The HTML format looks for a view in `app/views/posts/` with a name that corresponds to the action name. Rails makes all of the instance variables from the action available to the view. Here's `app/view/posts/index.html.erb`:

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
  </tr>

<% for post in @posts %>
  <tr>
    <td><%=h post.name %></td>
    <td><%=h post.title %></td>
    <td><%=h post.content %></td>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post) %></td>
    <td><%= link_to 'Destroy', post, :confirm => 'Are you sure?',
            :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New post', new_post_path %>
```

This view iterates over the contents of the `@posts` array to display content and links. A few things to note in the view:

- h is a Rails helper method to sanitize displayed data, preventing cross-site scripting attacks
- `link_to` builds a hyperlink to a particular destination
- `edit_post_path` is a helper that Rails provides as part of RESTful routing. You'll see a variety of these helpers for the different actions that the controller includes.

Customizing the Layout

The view is only part of the story of how HTML is displayed in your web browser. Rails also has the concept of `layouts`, which are containers for views. When Rails renders a view to the browser, it does so by putting the view's HTML into a layout's HTML. The `script/generate scaffold` command automatically created a default layout, `app/views/layouts/posts.html.erb`, for the posts. Open this layout in your editor and modify the body tag:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type"
        content="text/html;charset=UTF-8" />
  <title>Posts: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body style="background: #EEEEEE;">

<p style="color: green"><%= flash[:notice] %></p>

<%= yield  %>

</body>
</html>
```

Now when you refresh the `/posts` page, you'll see a gray background to the page. This same gray background will be used throughout all the views for posts.

Creating New Posts

Creating a new post involves two actions. The first is the new action, which instantiates an empty `Post` object:

```
def new
  @post = Post.new

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

The `new.html.erb` view displays this empty Post to the user:

```
<h1>New post</h1>

<% form_for(@post) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

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

The `form_for` block is used to create an HTML form. Within this block, you have access to methods to build various controls on the form. For example, `f.text_field :name` tells Rails to create a text input on the form, and to hook it up to the `name` attribute of the instance being displayed. You can only use these methods with attributes of the model that the form is based on (in this case `name`, `title`, and `content`). Rails uses `form_for` in preference to having your write raw HTML because the code is more succinct, and because it explicitly ties the form to a particular model instance.

If you need to create an HTML form that displays arbitrary fields, not tied to a model, you should use the `form_tag` method, which provides shortcuts for building forms that are not necessarily tied to a model instance.

When the user clicks the `Create` button on this form, the browser will send information back to the `create` method of the controller (Rails knows to call the `create` method because the form is sent with an HTTP POST request; that's one of the conventions that I mentioned earlier):

```
def create
  @post = Post.new(params[:post])

  respond_to do |format|
    if @post.save
      flash[:notice] = 'Post was successfully created.'
      format.html { redirect_to(@post) }
      format.xml  { render :xml => @post, :status => :created,
                      :location => @post }
    else
      format.html { render :action => "new" }
      format.xml  { render :xml => @post.errors,
                      :status => :unprocessable_entity }
    end
  end
end
```

The `create` action instantiates a new Post object from the data supplied by the user on the form, which Rails makes available in the `params` hash. After saving the new post, it uses `flash[:notice]` to create an informational message for the user, and redirects to the show action for the post. If there's any problem, the `create` action just shows the new view a second time, with any error messages.

Rails provides the `flash` hash (usually just called the Flash) so that messages can be carried over to another action, providing the user with useful information on the status of their request. In the case of `create`, the user never actually sees any page rendered during the Post creation process, because it immediately redirects to the new Post as soon Rails saves the record. The Flash carries over a message to the next action, so that when the user is redirected back to the show action, they are presented with a message saying "Post was successfully created."

Showing an Individual Post

When you click the show link for a post on the index page, it will bring you to a URL like `http://localhost:3000/posts/1`. Rails interprets this as a call to the show action for the resource, and passes in 1 as the `:id` parameter. Here's the show action:

```
def show
  @post = Post.find(params[:id])

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

The show action uses `Post.find` to search for a single record in the database by its id value. After finding the record, Rails displays it by using `show.html.erb`:

```
<p>
  <b>Name:</b>
  <%=h @post.name %>
</p>

<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>

<p>
  <b>Content:</b>
  <%=h @post.content %>
</p>


<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

Editing Posts

Like creating a new post, editing a post is a two-part process. The first step is a request to `edit_post_path(@post)` with a particular post. This calls the `edit` action in the controller:

```
def edit
  @post = Post.find(params[:id])
end
```

After finding the requested post, Rails uses the `edit.html.erb` view to display it:

```
<h1>Editing post</h1>

<% form_for(@post) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```

Submitting the form created by this view will invoke the `update` action within the controller:

```
def update
  @post = Post.find(params[:id])

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

In the `update` action, Rails first uses the `:id` parameter passed back from the edit view to locate the database record that's being edited. The `update_attributes` call then takes the rest of the parameters from the request and applies them to this record. If all goes well, the user is redirected to the post's `show` view. If there are any problems, it's back to `edit` to correct them.

Sharp-eyed readers will have noticed that the `form_for` declaration is identical for the new and `edit` views. Rails generates different code for the two forms because it's smart enough to notice that in the one case it's being passed a new record that has never been saved, and in the other case an existing record that has already been saved to the database. In a production Rails application, you would ordinarily eliminate this duplication by moving identical code to a *partial template*, which you could then include in both parent templates. But the scaffold generator tries not to make too many assumptions, and generates code that's easy to modify if you want different forms for `create` and `edit`.

Destroying a Post

Finally, clicking one of the `destroy` links sends the associated id to the `destroy` action:

```
def destroy
  @post = Post.find(params[:id])
  @post.destroy

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

The `destroy` method of an Active Record model instance removes the corresponding record from the database. After that's done, there isn't any record to display, so Rails redirects the user's browser to the index view for the model.

DRYing up the Code

At this point, it's worth looking at some of the tools that Rails provides to eliminate duplication in your code. In particular, you can use *partials* to clean up duplication in views and *filters* to help with duplication in controllers.

Using Partials to Eliminate View Duplication

As you saw earlier, the scaffold-generated views for the new and `edit` actions are largely identical. You can pull the shared code out into a partial template. This requires editing the new and edit views, and adding a new template. The new `_form.html.erb` template should be saved in the same `app/views/posts` folder as the files from which it is being extracted. Note that the name of this file begins with an underscore; that's the Rails naming convention for partial templates.

NEW.HTML.ERB

```
<h1>New post</h1>

<%= render :partial => "form" %>

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

EDIT.HTML.ERB

```
<h1>Editing post</h1>

<%= render :partial => "form" %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```

_FORM.HTML.ERB

```
<% form_for(@post) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :title, "title" %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Save" %>
  </p>
<% end %>
```

Now, when Rails renders the `new` or `edit` view, it will insert the `_form` partial at the indicated point. Note the naming convention for partials: if you refer to a partial named `form` inside of a view, the corresponding file is `_form.html.erb`, with a leading underscore.

Using Filters to Eliminate Controller Duplication

At this point, if you look at the controller for posts, you'll see some duplication:

```
class PostsController < ApplicationController
  # ...
  def show
    @post = Post.find(params[:id])
  # ...
  end

  def edit
    @post = Post.find(params[:id])
  end

  def update
    @post = Post.find(params[:id])
    # ...
  end

  def destroy
    @post = Post.find(params[:id])
    # ...
  end
end
```

Four instances of the exact same line of code doesn't seem very DRY. Rails provides *filters* as a way to address this sort of repeated code. In this case, you can DRY things up by using a `before_filter`:

```
class PostsController < ApplicationController
  before_filter :find_post,
    :only => [:show, :edit, :update, :destroy]
  # ...
  def show
  # ...
  end

  def edit
  end

  def update
    # ...
  end

  def destroy
    # ...
  end

  private
    def find_post
      @post = Post.find(params[:id])
    end
end
```

Rails runs *before filters* before any action in the controller. You can use the `:only` clause to limit a before filter to only certain actions, or an `:except` clause to specifically skip a before filter for certain actions. Rails also allows you to define *after filters* that run after processing an action, as well as *around filters* that surround the processing of actions. Filters can also be defined in external classes to make it easy to share them between controllers.

Adding a Second Model

Now that you've seen what's in a model built with scaffolding, it's time to add a second model to the application. The second model will handle comments on blog posts.

Generating a Model

Models in Rails use a singular name, and their corresponding database tables use a plural name. For the model to hold comments, the convention is to use the name Comment. Even if you don't want to use the entire apparatus set up by scaffolding, most Rails developers still use generators to make things like models and controllers. To create the new model, run this command in your terminal:

```
$ script/generate model Comment commenter:string body:text post:references
```

This command will generate four files:

- `app/models/comment.rb` – The model
- `db/migrate/20091013214407_create_comments.rb` – The migration
- `test/unit/comment_test.rb and test/fixtures/comments.yml` – The test harness.

First, take a look at `comment.rb`:

```
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

This is very similar to the `post.rb` model that you saw earlier. The difference is the line `belongs_to :post`, which sets up an Active Record *association*. You'll learn a little about associations in the next section of this guide.

In addition to the model, Rails has also made a migration to create the corresponding database table:

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
      t.string :commenter
      t.text :body
      t.references :post

      t.timestamps
    end
  end

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

The `t.references` line sets up a foreign key column for the association between the two models. Go ahead and run the migration:

```
$ rake db:migrate
```

Rails is smart enough to only execute the migrations that have not already been run against the current database.

Associating Models

Active Record associations let you easily declare the relationship between two models. In the case of comments and posts, you could write out the relationships this way:

- Each comment belongs to one post
- One post can have many comments

In fact, this is very close to the syntax that Rails uses to declare this association. You've already seen the line of code inside the Comment model that makes each comment belong to a Post:

```
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

You'll need to edit the `post.rb` file to add the other side of the association:

```
class Post < ActiveRecord::Base
  validates_presence_of :name, :title
  validates_length_of :title, :minimum => 5
  has_many :comments
end
```

These two declarations enable a good bit of automatic behavior. For example, if you have an instance variable `@post` containing a post, you can retrieve all the comments belonging to that post as the array `@post.comments`.

For more information on Active Record associations, see the Active Record Associations guide.

Adding a Route

Routes are entries in the `config/routes.rb` file that tell Rails how to match incoming HTTP requests to controller actions. Open up that file and find the existing line referring to `posts` (it will be right at the top of the file). Then edit it as follows:

```
map.resources :posts, :has_many => :comments
```

This creates `comments` as a *nested resource* within `posts`. This is another part of capturing the hierarchical relationship that exists between posts and comments.

For more information on routing, see the Rails Routing from the Outside In guide.

Generating a Controller

With the model in hand, you can turn your attention to creating a matching controller. Again, there's a generator for this:

```
$ script/generate controller Comments index show new edit
```

This creates seven files:

- `app/controllers/comments_controller.rb` – The controller
- `app/helpers/comments_helper.rb` – A view helper file
- `app/views/comments/index.html.erb` – The view for the index action
- `app/views/comments/show.html.erb` – The view for the show action
- `app/views/comments/new.html.erb` – The view for the new action
- `app/views/comments/edit.html.erb` – The view for the edit action
- `test/functional/comments_controller_test.rb` – The functional tests for the controller

The controller will be generated with empty methods and views for each action that you specified in the call to `script/generate controller`:

```
class CommentsController < ApplicationController
  def index
  end

  def show
  end

  def new
  end

  def edit
  end
end
```

You'll need to flesh this out with code to actually process requests appropriately in each method.

The next page has a version that (for simplicity's sake) only responds to requests that require HTML.

```
class CommentsController < ApplicationController
  def index
    @post = Post.find(params[:post_id])
    @comments = @post.comments
  end

  def show
    @post = Post.find(params[:post_id])
    @comment = @post.comments.find(params[:id])
  end

  def new
    @post = Post.find(params[:post_id])
    @comment = @post.comments.build
  end

  def create
    @post = Post.find(params[:post_id])
    @comment = @post.comments.build(params[:comment])
    if @comment.save
      redirect_to post_comment_url(@post, @comment)
    else
      render :action => "new"
    end
  end

  def edit
    @post = Post.find(params[:post_id])
    @comment = @post.comments.find(params[:id])
  end

  def update
    @post = Post.find(params[:post_id])
    @comment = Comment.find(params[:id])
    if @comment.update_attributes(params[:comment])
      redirect_to post_comment_url(@post, @comment)
    else
      render :action => "edit"
    end
  end

  def destroy
    @post = Post.find(params[:post_id])
    @comment = Comment.find(params[:id])
    @comment.destroy

    respond_to do |format|
      format.html { redirect_to post_comments_path(@post) }
      format.xml  { head :ok }
    end
  end
end
```

You'll see a bit more complexity here than you did in the controller for posts. That's a side-effect of the nesting that you've set up; each request for a comment has to keep track of the post to which the comment is attached.

In addition, the code takes advantage of some of the methods available for an association. For example, in the `new` method, it calls

```
@comment = @post.comments.build
```

This creates a new `Comment` object *and* sets up the `post_id` field to have the `id` from the specified `Post` object in a single operation.

Building Views

Because you skipped scaffolding, you'll need to build views for comments "by hand." Invoking `script/generate controller` will give you skeleton views, but they'll be devoid of actual content. Here's a first pass at fleshing out the comment views.

VIEWS/COMMENTS/INDEX.HTML.ERB

```
<h1>Comments for <%= @post.title %></h1>

<table>
  <tr>
    <th>Commenter</th>
    <th>Body</th>
  </tr>

<% for comment in @comments %>
  <tr>
    <td><%=h comment.commenter %></td>
    <td><%=h comment.body %></td>
    <td><%= link_to 'Show', post_comment_path(@post, comment) %></td>
    <td>
        <%= link_to 'Edit', edit_post_comment_path(@post, comment) %>
    </td>
    <td>
        <%= link_to 'Destroy', post_comment_path(@post, comment),
            :confirm => 'Are you sure?', :method => :delete %>
    </td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New comment', new_post_comment_path(@post) %>
<%= link_to 'Back to Post', @post %>
```

VIEWS/COMMENTS/NEW.HTML.ERB

```
<h1>New comment</h1>

<% form_for([@post, @comment]) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

<%= link_to 'Back', post_comments_path(@post) %>
```

VIEWS/COMMENTS/SHOW.HTML.ERB

```
<h1>Comment on <%= @post.title %></h1>

<p>
  <b>Commenter:</b>
  <%=h @comment.commenter %>
</p>

<p>
  <b>Comment:</b>
  <%=h @comment.body %>
</p>

<%= link_to 'Edit', edit_post_comment_path(@post, @comment) %> |
<%= link_to 'Back', post_comments_path(@post) %>
```

VIEWS/COMMENTS/EDIT.HTML.ERB

```
<h1>Editing comment</h1>

<% form_for([@post, @comment]) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', post_comment_path(@post, @comment) %> |
<%= link_to 'Back', post_comments_path(@post) %>
```

Again, the added complexity here (compared to the views you saw for managing posts) comes from the necessity of juggling a post and its comments at the same time.

Hooking Comments to Posts

As a next step, I'll modify the `views/posts/show.html.erb` view to show the comments on that post, and to allow managing those comments:

```
<p>
  <b>Name:</b>
  <%=h @post.name %>
</p>

<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>

<p>
  <b>Content:</b>
  <%=h @post.content %>
</p>

<h2>Comments</h2>
<% @post.comments.each do |c| %>
  <p>
    <b>Commenter:</b>
    <%=h c.commenter %>
  </p>

  <p>
    <b>Comment:</b>
    <%=h c.body %>
  </p>
<% end %>

<%= link_to 'Edit Post', edit_post_path(@post) %> |
<%= link_to 'Back to Posts', posts_path %> |
<%= link_to 'Manage Comments', post_comments_path(@post) %>
```

Note that each post has its own individual comments collection, accessible as `@post.comments`. That's a consequence of the declarative associations in the models. Path helpers such as `post_comments_path` come from the nested route declaration in `config/routes.rb`.

Building a Multi-Model Form

Comments and posts are edited on two separate forms – which makes sense, given the flow of this mini-application. But what if you want to edit more than one thing on a single form? Rails 2.3 offers new support for nested forms. Let's add support for giving each post multiple tags, right in the form where you create the post. First, create a new model to hold the tags:

```
$ script/generate model tag name:string post:references
```

Run the migration to create the database table:

```
$ rake db:migrate
```

Next, edit the `post.rb` file to create the other side of the association, and to tell Rails that you intend to edit tags via posts:

```
class Post < ActiveRecord::Base
  validates_presence_of :name, :title
  validates_length_of :title, :minimum => 5
  has_many :comments
  has_many :tags

  accepts_nested_attributes_for :tags, :allow_destroy => :true  ,
    :reject_if => proc { |attrs| attrs.all? { |k, v| v.blank? } }
end
```

The `:allow_destroy` option on the nested attribute declaration tells Rails to display a “remove” checkbox on the view that you’ll build shortly. The `:reject_if` option prevents saving new tags that do not have any attributes filled in.

You'll also need to modify `views/posts/_form.html.erb` to include the tags:

```
<% @post.tags.build if @post.tags.empty? %>
<% form_for(@post) do |post_form| %>
  <%= post_form.error_messages %>

  <p>
    <%= post_form.label :name %><br />
    <%= post_form.text_field :name %>
  </p>
  <p>
    <%= post_form.label :title, "title" %><br />
    <%= post_form.text_field :title %>
  </p>
  <p>
    <%= post_form.label :content %><br />
    <%= post_form.text_area :content %>
  </p>
  <h2>Tags</h2>
  <% post_form.fields_for :tags do |tag_form| %>
    <p>
      <%= tag_form.label :name, 'Tag:' %>
      <%= tag_form.text_field :name %>
    </p>
    <% unless tag_form.object.nil? || tag_form.object.new_record? %>
      <p>
        <%= tag_form.label :_delete, 'Remove:' %>
        <%= tag_form.check_box :_delete %>
      </p>
    <% end %>
  <% end %>

  <p>
    <%= post_form.submit "Save" %>
  </p>
<% end %>
```

With these changes in place, you'll find that you can edit a post and its tags directly on the same view.