**More at**[**rubyonrails.org:**](http://rubyonrails.org/)[Overview](http://rubyonrails.org/) | [Download](http://rubyonrails.org/download) | [Deploy](http://rubyonrails.org/deploy) | [Code](https://github.com/rails/rails) | [Screencasts](http://rubyonrails.org/screencasts) | [Documentation](http://rubyonrails.org/documentation) | [Ecosystem](http://rubyonrails.org/ecosystem) | [Community](http://rubyonrails.org/community) | [Blog](http://weblog.rubyonrails.org/)

[**Guides.rubyonrails.org**](http://guides.rubyonrails.org/v3.2.13/index.html)

* [Home](http://guides.rubyonrails.org/v3.2.13/index.html)

* [Guides Index](http://guides.rubyonrails.org/v3.2.13/index.html)
* [Contribute](http://guides.rubyonrails.org/v3.2.13/contributing_to_ruby_on_rails.html)

* [Credits](http://guides.rubyonrails.org/v3.2.13/credits.html)

**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**

**Chapters**

1. [**Guide Assumptions**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#guide-assumptions)
2. [**What is Rails?**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#what-is-rails)
   * [The MVC Architecture](http://guides.rubyonrails.org/v3.2.13/getting_started.html#the-mvc-architecture)
   * [The Components of Rails](http://guides.rubyonrails.org/v3.2.13/getting_started.html#the-components-of-rails)
   * [REST](http://guides.rubyonrails.org/v3.2.13/getting_started.html#rest)
3. [**Creating a New Rails Project**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#creating-a-new-rails-project)
   * [Installing Rails](http://guides.rubyonrails.org/v3.2.13/getting_started.html#installing-rails)
   * [Creating the Blog Application](http://guides.rubyonrails.org/v3.2.13/getting_started.html#creating-the-blog-application)
   * [Configuring a Database](http://guides.rubyonrails.org/v3.2.13/getting_started.html#configuring-a-database)
   * [Creating the Database](http://guides.rubyonrails.org/v3.2.13/getting_started.html#creating-the-database)
4. [**Hello, Rails!**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#hello-rails)
   * [Starting up the Web Server](http://guides.rubyonrails.org/v3.2.13/getting_started.html#starting-up-the-web-server)
   * [Say “Hello”, Rails](http://guides.rubyonrails.org/v3.2.13/getting_started.html#say-hello-rails)
   * [Setting the Application Home Page](http://guides.rubyonrails.org/v3.2.13/getting_started.html#setting-the-application-home-page)
5. [**Getting Up and Running Quickly with Scaffolding**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#getting-up-and-running-quickly-with-scaffolding)
6. [**Creating a Resource**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#creating-a-resource)
   * [Running a Migration](http://guides.rubyonrails.org/v3.2.13/getting_started.html#running-a-migration)
   * [Adding a Link](http://guides.rubyonrails.org/v3.2.13/getting_started.html#adding-a-link)
   * [Working with Posts in the Browser](http://guides.rubyonrails.org/v3.2.13/getting_started.html#working-with-posts-in-the-browser)
   * [The Model](http://guides.rubyonrails.org/v3.2.13/getting_started.html#the-model)
   * [Adding Some Validation](http://guides.rubyonrails.org/v3.2.13/getting_started.html#adding-some-validation)
   * [Using the Console](http://guides.rubyonrails.org/v3.2.13/getting_started.html#using-the-console)
   * [Listing All Posts](http://guides.rubyonrails.org/v3.2.13/getting_started.html#listing-all-posts)
   * [Customizing the Layout](http://guides.rubyonrails.org/v3.2.13/getting_started.html#customizing-the-layout)
   * [Creating New Posts](http://guides.rubyonrails.org/v3.2.13/getting_started.html#creating-new-posts)
   * [Showing an Individual Post](http://guides.rubyonrails.org/v3.2.13/getting_started.html#showing-an-individual-post)
   * [Editing Posts](http://guides.rubyonrails.org/v3.2.13/getting_started.html#editing-posts)
   * [Destroying a Post](http://guides.rubyonrails.org/v3.2.13/getting_started.html#destroying-a-post)
7. [**Adding a Second Model**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#adding-a-second-model)
   * [Generating a Model](http://guides.rubyonrails.org/v3.2.13/getting_started.html#generating-a-model)
   * [Associating Models](http://guides.rubyonrails.org/v3.2.13/getting_started.html#associating-models)
   * [Adding a Route for Comments](http://guides.rubyonrails.org/v3.2.13/getting_started.html#adding-a-route-for-comments)
   * [Generating a Controller](http://guides.rubyonrails.org/v3.2.13/getting_started.html#generating-a-controller)
8. [**Refactoring**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#refactoring)
   * [Rendering Partial Collections](http://guides.rubyonrails.org/v3.2.13/getting_started.html#rendering-partial-collections)
   * [Rendering a Partial Form](http://guides.rubyonrails.org/v3.2.13/getting_started.html#rendering-a-partial-form)
9. [**Deleting Comments**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#deleting-comments)
   * [Deleting Associated Objects](http://guides.rubyonrails.org/v3.2.13/getting_started.html#deleting-associated-objects)
10. [**Security**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#security)
11. [**Building a Multi-Model Form**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#building-a-multi-model-form)
12. [**View Helpers**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#view-helpers)
13. [**What’s Next?**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#what-s-next)
14. [**Configuration Gotchas**](http://guides.rubyonrails.org/v3.2.13/getting_started.html#configuration-gotchas)

This Guide is based on Rails 3.2. Some of the code shown here will not work in earlier versions of Rails.

**1 Guide Assumptions**

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](http://www.ruby-lang.org/en/downloads) language version 1.8.7 or higher

Note that Ruby 1.8.7 p248 and p249 have marshaling bugs that crash Rails 3.0 and above. Ruby Enterprise Edition have these fixed since release 1.8.7-2010.02 though. On the 1.9 front, Ruby 1.9.1 is not usable because it outright segfaults on Rails 3.0 and above, so if you want to use Rails 3.0 or above with 1.9.x jump on 1.9.2 for smooth sailing.

* The [RubyGems](http://rubyforge.org/frs/?group_id=126) packaging system
  + If you want to learn more about RubyGems, please read the [RubyGems User Guide](http://docs.rubygems.org/read/book/1)
* A working installation of the [SQLite3 Database](http://www.sqlite.org/)

Rails is a web application framework running on the Ruby programming language. If you have no prior experience with Ruby, you will find a very steep learning curve diving straight into Rails. There are some good free resources on the internet for learning Ruby, including:

* [Mr. Neighborly’s Humble Little Ruby Book](http://www.humblelittlerubybook.com/)
* [Programming Ruby](http://www.ruby-doc.org/docs/ProgrammingRuby/)
* [Why’s (Poignant) Guide to Ruby](http://mislav.uniqpath.com/poignant-guide/)

Also, the example code for this guide is available in the rails github:https://github.com/rails/rails repository in rails/railties/guides/code/getting\_started.

**2 What is Rails?**

This section goes into the background and philosophy of the Rails framework in detail. You can safely skip this section and come back to it at a later time. Section 3 starts you on the path to creating your first Rails application.

Rails is a web application development framework written in the Ruby language. It is designed to make programming web applications easier by making 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. Experienced Rails developers also report that it makes web application development more fun.

Rails is opinionated software. It makes the assumption that there is a “best” way to do things, and it’s designed to encourage that 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 requiring you to specify 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.

**2.1 The MVC Architecture**

At the core of Rails is the Model, View, Controller architecture, usually just called MVC. MVCbenefits 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

**2.1.1 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, each 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.

**2.1.2 Views**

Views represent the user interface of your application. In Rails, views are often HTML files with embedded Ruby code that perform 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.

**2.1.3 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.

**2.2 The Components of Rails**

Rails ships as many individual components. Each of these components are briefly explained below. If you are new to Rails, as you read this section, don’t get hung up on the details of each component, as they will be explained in further detail later. For instance, we will bring up Rack applications, but you don’t need to know anything about them to continue with this guide.

* Action Pack
  + Action Controller
  + Action Dispatch
  + Action View
* Action Mailer
* Active Model
* Active Record
* Active Resource
* Active Support
* Railties

**2.2.1 Action Pack**

Action Pack is a single gem that contains Action Controller, Action View and Action Dispatch. The “VC” part of “MVC”.

2.2.1.1 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.

2.2.1.2 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. View templates are covered in more detail in another guide called[Layouts and Rendering](http://guides.rubyonrails.org/v3.2.13/layouts_and_rendering.html).

2.2.1.3 Action Dispatch

Action Dispatch handles routing of web requests and dispatches them as you want, either to your application or any other Rack application. Rack applications are a more advanced topic and are covered in a separate guide called [Rails on Rack](http://guides.rubyonrails.org/v3.2.13/rails_on_rack.html).

**2.2.2 Action Mailer**

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

**2.2.3 Active Model**

Active Model provides a defined interface between the Action Pack gem services and Object Relationship Mapping gems such as Active Record. Active Model allows Rails to utilize other ORMframeworks in place of Active Record if your application needs this.

**2.2.4 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.

**2.2.5 Active Resource**

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

**2.2.6 Active Support**

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

**2.2.7 Railties**

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

**2.3 REST**

Rest stands for Representational State Transfer and is the foundation of the RESTful architecture. This is generally considered to be Roy Fielding’s doctoral thesis, [Architectural Styles and the Design of Network-based Software Architectures](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). While you can read through the thesis, REST in terms of Rails boils down to two main principles:

* Using resource identifiers such as URLs to represent resources.
* Transferring representations of the state of that resource between system components.

For example, the following HTTP request:

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 hooks into this shielding you from many of the RESTful complexities and browser quirks.

If you’d like more details on REST as an architectural style, these resources are more approachable than Fielding’s thesis:

* [A Brief Introduction to REST](http://www.infoq.com/articles/rest-introduction) by Stefan Tilkov
* [An Introduction to REST](http://bitworking.org/news/373/An-Introduction-to-REST) (video tutorial) by Joe Gregorio
* [Representational State Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer) article in Wikipedia
* [How to GET a Cup of Coffee](http://www.infoq.com/articles/webber-rest-workflow) by Jim Webber, Savas Parastatidis & Ian Robinson

**3 Creating a New Rails Project**

The best way to use this guide is to follow each step as it happens, no code or step needed to make this example application has been left out, so you can literally follow along step by step. You can get the complete code [here](https://github.com/rails/rails/tree/3-2-stable/railties/guides/code/getting_started).

By following along with 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.

The examples below use # and $ to denote terminal prompts. If you are using Windows, your prompt will look something like c:\source\_code>

**3.1 Installing Rails**

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

|  |
| --- |
| Usually run this as the root user:  # gem install rails |

If you’re working on Windows, you can quickly install Ruby and Rails with [Rails Installer](http://railsinstaller.org/).

To verify that you have everything installed correctly, you should be able to run the following:

|  |
| --- |
| $ rails --version |

If it says something like “Rails 3.2.3” you are ready to continue.

**3.2 Creating the Blog Application**

To begin, open a terminal, navigate to a folder where you have rights to create files, and type:

|  |
| --- |
| $ rails new blog |

This will create a Rails application called Blog in a directory called blog.

You can see all of the switches that the Rails application builder accepts by runningrails new -h.

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

|  |
| --- |
| $ cd blog |

The ‘rails new blog’ command we ran above created a folder in your working directory called blog. The blog folder has a number of auto-generated folders that make up the structure of a Rails application. Most of the work in this tutorial will happen in the app/ folder, but here’s a basic rundown on the function of each of the files and folders that Rails created by default:

|  |  |
| --- | --- |
| **File/Folder** | **Purpose** |
| app/ | Contains the controllers, models, views and assets 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. This is covered in more detail in [Configuring Rails Applications](http://guides.rubyonrails.org/v3.2.13/configuring.html) |
| config.ru | Rack configuration for Rack based servers used to start the application. |
| db/ | Contains your current database schema, as well as the database migrations. |
| doc/ | In-depth documentation for your application. |
| Gemfile Gemfile.lock | These files allow you to specify what gem dependencies are needed for your Rails application. |
| lib/ | Extended modules for your application. |
| log/ | Application log files. |
| public/ | The only folder seen to the world as-is. Contains the static files and compiled assets. |
| Rakefile | This file locates and loads tasks that can be run from the command line. The task definitions are defined throughout the components of Rails. Rather than changing Rakefile, you should add your own tasks by adding files to the lib/tasks directory of your application. |
| README.rdoc | This is a brief instruction manual for your application. You should edit this file to tell others what your application does, how to set it up, and so on. |
| script/ | Contains the rails script that starts your app and can contain other scripts you use to deploy or run your application. |
| test/ | Unit tests, fixtures, and other test apparatus. These are covered in [Testing Rails Applications](http://guides.rubyonrails.org/v3.2.13/testing.html) |
| tmp/ | Temporary files |
| vendor/ | A place for all third-party code. In a typical Rails application, this includes Ruby Gems, the Rails source code (if you optionally install it into your project) and plugins containing additional prepackaged functionality. |

**3.3 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 configured to use SQLite3. The file contains sections for three different environments in which Rails can run by default:

* The development environment is used on your development/local computer as you interact manually with the application.
* The test environment is used when running automated tests.
* The production environment is used when you deploy your application for the world to use.

You don’t have to update the database configurations manually. If you look at the options of the application generator, you will see that one of the options is named —database. This option allows you to choose an adapter from a list of the most used relational databases. You can even run the generator repeatedly: cd .. && rails new blog —database=mysql. When you confirm the overwriting of the config/database.yml file, your application will be configured for MySQL instead of SQLite. Detailed examples of the common database connections are below.

**3.3.1 Configuring an SQLite3 Database**

Rails comes with built-in support for [SQLite3](http://www.sqlite.org/), 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 an SQLite database when creating a new project, but you can always change it later.

Here’s the section of the default configuration file (config/database.yml) with connection information for the development environment:

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

In this guide we are using an SQLite3 database for data storage, because it is a zero configuration database that just works. Rails also supports MySQL and PostgreSQL “out of the box”, and has plugins for many database systems. If you are using a database in a production environment Rails most likely has an adapter for it.

**3.3.2 Configuring a MySQL Database**

If you choose to use MySQL instead of the shipped SQLite3 database, yourconfig/database.yml will look a little different. Here’s the development section:

|  |
| --- |
| development:    adapter: mysql2    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 thedevelopment section as appropriate.

**3.3.3 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: |

**3.3.4 Configuring an SQLite3 Database for JRuby Platform**

If you choose to use SQLite3 and are using JRuby, your config/database.yml will look a little different. Here’s the development section:

|  |
| --- |
| development:    adapter: jdbcsqlite3    database: db/development.sqlite3 |

**3.3.5 Configuring a MySQL Database for JRuby Platform**

If you choose to use MySQL and are using JRuby, your config/database.yml will look a little different. Here’s the development section:

|  |
| --- |
| development:    adapter: jdbcmysql    database: blog\_development    username: root    password: |

**3.3.6 Configuring a PostgreSQL Database for JRuby Platform**

Finally if you choose to use PostgreSQL and are using JRuby, your config/database.yml will look a little different. Here’s the development section:

|  |
| --- |
| development:    adapter: jdbcpostgresql    encoding: unicode    database: blog\_development    username: blog    password: |

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

**3.4 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 |

This will create your development and test SQLite3 databases inside the db/ folder.

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.

**4 Hello, Rails!**

One of the traditional places to start with a new language is by getting some text up on screen quickly. To do this, you need to get your Rails application server running.

**4.1 Starting up the Web Server**

You actually have a functional Rails application already. To see it, you need to start a web server on your development machine. You can do this by running:

|  |
| --- |
| $ rails server |

Compiling CoffeeScript to JavaScript requires a JavaScript runtime and the absence of a runtime will give you an execjs error. Usually Mac OS X and Windows come with a JavaScript runtime installed. Rails adds the therubyracer gem to Gemfile in a commented line for new apps and you can uncomment if you need it. therubyrhino is the recommended runtime for JRuby users and is added by default to Gemfile in apps generated under JRuby. You can investigate about all the supported runtimes at [ExecJS](https://github.com/sstephenson/execjs#readme).

This will fire up an instance of the WEBrick 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](http://localhost:3000/). You should see Rails’ default information page:

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. You can also click on the *About your application’s environment* link to see a summary of your application’s environment.

**4.2 Say “Hello”, Rails**

To get Rails saying “Hello”, 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:

|  |
| --- |
| $ rails generate controller home index |

If you get a command not found error when running this command, you need to explicitly pass Rails rails commands to Ruby: ruby \path\to\your\application\script\rails 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>

**4.3 Setting the Application Home Page**

Now that we have made the controller and view, we need to tell Rails when we want “Hello Rails!” to show up. In our case, we want it to show up when we navigate to the root URL of our site,[http://localhost:3000](http://localhost:3000/), instead of the “Welcome Aboard” smoke test.

The first step to doing this is to delete the default page from your application:

|  |
| --- |
| $ rm public/index.html |

We need to do this as Rails will deliver any static file in the public directory in preference to any dynamic content we generate from the controllers.

Now, you have to tell Rails where your actual home page is located. Open the fileconfig/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. This file contains many sample routes on commented lines, and one of them actually shows you how to connect the root of your site to a specific controller and action. Find the line beginning with root :to and uncomment it. It should look something like the following:

|  |
| --- |
| Blog::Application.routes.draw do      #...    # You can have the root of your site routed with "root"    # just remember to delete public/index.html.    root :to => "home#index" |

The root :to => "home#index" tells Rails to map the root action to the home controller’s index action.

Now if you navigate to [http://localhost:3000](http://localhost:3000/) in your browser, you’ll see Hello, Rails!.

For more information about routing, refer to [Rails Routing from the Outside In](http://guides.rubyonrails.org/v3.2.13/routing.html).

**5 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.

**6 Creating a Resource**

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

|  |
| --- |
| $ rails generate scaffold Post name:string title:string content:text |

The scaffold generator will build several files in your application, along with some folders, and editconfig/routes.rb. Here’s a quick overview of what it creates:

|  |  |
| --- | --- |
| **File** | **Purpose** |
| db/migrate/20100207214725\_create\_posts.rb | Migration to create the posts table in your database (your name will include a different timestamp) |
| app/models/post.rb | The Post model |
| test/unit/post\_test.rb | Unit testing harness for the posts model |
| test/fixtures/posts.yml | Sample posts for use in testing |
| config/routes.rb | Edited to include routing information for posts |
| app/controllers/posts\_controller.rb | The Posts controller |
| app/views/posts/index.html.erb | A view to display an index of all posts |
| app/views/posts/edit.html.erb | A view to edit an existing post |
| 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/\_form.html.erb | A partial to control the overall look and feel of the form used in edit and new views |
| 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 post views |
| test/unit/helpers/posts\_helper\_test.rb | Unit testing harness for the posts helper |
| app/assets/javascripts/posts.js.coffee | CoffeeScript for the posts controller |
| app/assets/stylesheets/posts.css.scss | Cascading style sheet for the posts controller |
| app/assets/stylesheets/scaffolds.css.scss | Cascading style sheet to make the scaffolded views look better |

While scaffolding will get you up and running quickly, the code it generates is unlikely to be a perfect fit for your application. You’ll most probably want to customize the generated code. Many experienced Rails developers avoid scaffolding entirely, preferring to write all or most of their source code from scratch. Rails, however, makes it really simple to customize templates for generated models, controllers, views and other source files. You’ll find more information in the [Creating and Customizing Rails Generators & Templates](http://guides.rubyonrails.org/v3.2.13/generators.html) guide.

**6.1 Running a Migration**

One of the products of the rails 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/20100207214725\_create\_posts.rb file (remember, yours will have a slightly different name), here’s what you’ll find:

|  |
| --- |
| class CreatePosts < ActiveRecord::Migration    def change      create\_table :posts do |t|        t.string :name        t.string :title        t.text :content          t.timestamps      end    end  end |

The above migration creates a method named change which will be called when you run this migration. The action defined in this method is also reversible, which means Rails knows how to reverse the change made by this migration, in case you want to reverse it later. When you run this migration it will create a posts table with two string columns and a text column. It also creates two timestamp fields to allow Rails to track post creation and update times. More information about Rails migrations can be found in the [Rails Database Migrations](http://guides.rubyonrails.org/v3.2.13/migrations.html) guide.

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

|  |
| --- |
| $ rake db:migrate |

Rails will execute this migration command and tell you it created the Posts table.

|  |
| --- |
| ==  CreatePosts: migrating ====================================================  -- create\_table(:posts)     -> 0.0019s  ==  CreatePosts: migrated (0.0020s) =========================================== |

Because you’re working in the development environment by default, this command will apply to the database defined in the development section of yourconfig/database.yml file. If you would like to execute migrations in another environment, for instance in production, you must explicitly pass it when invoking the command: rake db:migrate RAILS\_ENV=production.

**6.2 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.

**6.3 Working with Posts in the Browser**

Now you’re ready to start working with posts. To do that, navigate to [http://localhost:3000](http://localhost:3000/) and then click the “My Blog” link:

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 rails generate scaffold command.

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.

**6.4 The Model**

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

|  |
| --- |
| class Post < ActiveRecord::Base    attr\_accessible :content, :name, :title  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. Another important part of this file is attr\_accessible. It specifies a whitelist of attributes that are allowed to be updated in bulk (via update\_attributes for instance).

**6.5 Adding Some Validation**

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

|  |
| --- |
| class Post < ActiveRecord::Base    attr\_accessible :content, :name, :title      validates :name,  :presence => true    validates :title, :presence => true,                      :length => { :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. Validations are covered in detail in [Active Record Validations and Callbacks](http://guides.rubyonrails.org/v3.2.13/active_record_validations_callbacks.html#validations-overview)

**6.6 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:

|  |
| --- |
| $ rails console |

The default console will make changes to your database. You can instead open a console that will roll back any changes you make by using rails console —sandbox.

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

|  |
| --- |
| >> p = Post.new(: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.full\_messages  => ["Name can't be blank", "Title can't be blank", "Title 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.

When you’re finished, type exit and hit return to exit the console.

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

**6.7 Listing All Posts**

Let’s dive into the Rails code a little deeper to see how the application is showing us the list of Posts. Open the file app/controllers/posts\_controller.rb and look at the index action:

|  |
| --- |
| def index    @posts = Post.all      respond\_to do |format|      format.html  # index.html.erb      format.json  { render :json => @posts }    end  end |

Post.all returns all of the posts currently in the database as an array of Post records that we store in an instance variable called @posts.

For more information on finding records with Active Record, see [Active Record Query Interface](http://guides.rubyonrails.org/v3.2.13/active_record_querying.html).

The respond\_to block handles both HTML and JSON calls to this action. If you browse to<http://localhost:3000/posts.json>, you’ll see a JSON containing all of the posts. 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’sapp/views/posts/index.html.erb:

|  |
| --- |
| <h1>Listing posts</h1>    <table>    <tr>      <th>Name</th>      <th>Title</th>      <th>Content</th>      <th></th>      <th></th>      <th></th>    </tr>    <% @posts.each do |post| %>    <tr>      <td><%= post.name %></td>      <td><%= post.title %></td>      <td><%= 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:

* link\_to builds a hyperlink to a particular destination
* edit\_post\_path and new\_post\_path are helpers that Rails provides as part of RESTful routing. You’ll see a variety of these helpers for the different actions that the controller includes.

In previous versions of Rails, you had to use <%=h post.name %> so that any HTMLwould be escaped before being inserted into the page. In Rails 3 and above, this is now the default. To get unescaped HTML, you now use <%= raw post.name %>.

For more details on the rendering process, see [Layouts and Rendering in Rails](http://guides.rubyonrails.org/v3.2.13/layouts_and_rendering.html).

**6.8 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. In previous versions of Rails, the rails generate scaffold command would automatically create a controller specific layout, likeapp/views/layouts/posts.html.erb, for the posts controller. However this has been changed in Rails 3. An application specific layout is used for all the controllers and can be found inapp/views/layouts/application.html.erb. Open this layout in your editor and modify thebody tag to include the style directive below:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>    <title>Blog</title>    <%= stylesheet\_link\_tag "application" %>    <%= javascript\_include\_tag "application" %>    <%= csrf\_meta\_tags %>  </head>  <body style="background: #EEEEEE;">    <%= 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.

**6.9 Creating New Posts**

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

|  |
| --- |
| def new    @post = Post.new      respond\_to do |format|      format.html  # new.html.erb      format.json  { render :json => @post }    end  end |

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

|  |
| --- |
| <h1>New post</h1>    <%= render 'form' %>    <%= link\_to 'Back', posts\_path %> |

The <%= render 'form' %> line is our first introduction to *partials* in Rails. A partial is a snippet ofHTML and Ruby code that can be reused in multiple locations. In this case, the form used to make a new post is basically identical to the form used to edit a post, both having text fields for the name and title, a text area for the content, and a button to create the new post or to update the existing one.

If you take a look at views/posts/\_form.html.erb file, you will see the following:

|  |
| --- |
| <%= form\_for(@post) do |f| %>    <% if @post.errors.any? %>    <div id="errorExplanation">      <h2><%= pluralize(@post.errors.count, "error") %> prohibited          this post from being saved:</h2>      <ul>      <% @post.errors.full\_messages.each do |msg| %>        <li><%= msg %></li>      <% end %>      </ul>    </div>    <% end %>      <div class="field">      <%= f.label :name %><br />      <%= f.text\_field :name %>    </div>    <div class="field">      <%= f.label :title %><br />      <%= f.text\_field :title %>    </div>    <div class="field">      <%= f.label :content %><br />      <%= f.text\_area :content %>    </div>    <div class="actions">      <%= f.submit %>    </div>  <% end %> |

This partial receives all the instance variables defined in the calling view file. In this case, the controller assigned the new Post object to @post, which will thus be available in both the view and the partial as @post.

For more information on partials, refer to the [Layouts and Rendering in Rails](http://guides.rubyonrails.org/v3.2.13/layouts_and_rendering.html#using-partials) guide.

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 you write raw HTML because the code is more succinct, and because it explicitly ties the form to a particular model instance.

The form\_for block is also smart enough to work out if you are doing a *New Post* or an *Edit Post*action, and will set the form action tags and submit button names appropriately in the HTMLoutput.

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 Post button on this form, the browser will send information back to the create action of the controller (Rails knows to call the create action because the form is sent with an HTTP POST request; that’s one of the conventions that were mentioned earlier):

|  |
| --- |
| def create    @post = Post.new(params[:post])      respond\_to do |format|      if @post.save        format.html  { redirect\_to(@post,                      :notice => 'Post was successfully created.') }        format.json  { render :json => @post,                      :status => :created, :location => @post }      else        format.html  { render :action => "new" }        format.json  { render :json => @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 successfully saving the new post, createreturns the appropriate format that the user has requested (HTML in our case). It then redirects the user to the resulting post show action and sets a notice to the user that the Post was successfully created.

If the post was not successfully saved, due to a validation error, then the controller returns the user back to the new action with any error messages so that the user has the chance to fix the error and try again.

The “Post was successfully created.” message is stored in the Rails 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 Postas soon as 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.”

**6.10 Showing an Individual Post**

When you click the show link for a post on the index page, it will bring you to a URL likehttp://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.json  { render :json => @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 app/views/posts/show.html.erb:

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>      <%= link\_to 'Edit', edit\_post\_path(@post) %> |  <%= link\_to 'Back', posts\_path %> |

**6.11 Editing Posts**

Like creating a new post, editing a post is a two-part process. The first step is a request toedit\_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>    <%= render 'form' %>    <%= link\_to 'Show', @post %> |  <%= link\_to 'Back', posts\_path %> |

Again, as with the new action, the edit action is using the form partial. This time, however, the form will do a PUT action to the PostsController and the submit button will display “Update Post”.

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])        format.html  { redirect\_to(@post,                      :notice => 'Post was successfully updated.') }        format.json  { head :no\_content }      else        format.html  { render :action => "edit" }        format.json  { render :json => @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 postparameter (a hash) from the request and applies it to this record. If all goes well, the user is redirected to the post’s show action. If there are any problems, it redirects back to the edit action to correct them.

**6.12 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.json { head :no\_content }    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 action of the controller.

**7 Adding a Second Model**

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

**7.1 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:

|  |
| --- |
| $ rails generate model Comment commenter:string body:text post:references |

This command will generate four files:

|  |  |
| --- | --- |
| **File** | **Purpose** |
| db/migrate/20100207235629\_create\_comments.rb | Migration to create the comments table in your database (your name will include a different timestamp) |
| app/models/comment.rb | The Comment model |
| test/unit/comment\_test.rb | Unit testing harness for the comments model |
| test/fixtures/comments.yml | Sample comments for use in testing |

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 linebelongs\_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 change      create\_table :comments do |t|        t.string :commenter        t.text :body        t.references :post          t.timestamps      end        add\_index :comments, :post\_id    end  end |

The t.references line sets up a foreign key column for the association between the two models. And the add\_index line sets up an index for this association column. 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, so in this case you will just see:

|  |
| --- |
| ==  CreateComments: migrating =================================================  -- create\_table(:comments)     -> 0.0008s  -- add\_index(:comments, :post\_id)     -> 0.0003s  ==  CreateComments: migrated (0.0012s) ======================================== |

**7.2 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    attr\_accessible :content, :name, :title      validates :name,  :presence => true    validates :title, :presence => true,                      :length => { :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 an array using @post.comments.

For more information on Active Record associations, see the [Active Record Associations](http://guides.rubyonrails.org/v3.2.13/association_basics.html)guide.

**7.3 Adding a Route for Comments**

As with the home controller, we will need to add a route so that Rails knows where we would like to navigate to see comments. Open up the config/routes.rb file again. Near the top, you will see the entry for posts that was added automatically by the scaffold generator: resources :posts. Edit it as follows:

|  |
| --- |
| resources :posts do    resources :comments  end |

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](http://guides.rubyonrails.org/v3.2.13/routing.html) guide.

**7.4 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:

|  |
| --- |
| $ rails generate controller Comments |

This creates six files and one empty directory:

|  |  |
| --- | --- |
| **File/Directory** | **Purpose** |
| app/controllers/comments\_controller.rb | The Comments controller |
| app/views/comments/ | Views of the controller are stored here |
| test/functional/comments\_controller\_test.rb | The functional tests for the controller |
| app/helpers/comments\_helper.rb | A view helper file |
| test/unit/helpers/comments\_helper\_test.rb | The unit tests for the helper |
| app/assets/javascripts/comment.js.coffee | CoffeeScript for the controller |
| app/assets/stylesheets/comment.css.scss | Cascading style sheet for the controller |

Like with any blog, our readers will create their comments directly after reading the post, and once they have added their comment, will be sent back to the post show page to see their comment now listed. Due to this, our CommentsController is there to provide a method to create comments and delete spam comments when they arrive.

So first, we’ll wire up the Post show template (/app/views/posts/show.html.erb) to let us make a new comment:

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>    <h2>Add a comment:</h2>  <%= form\_for([@post, @post.comments.build]) do |f| %>    <div class="field">      <%= f.label :commenter %><br />      <%= f.text\_field :commenter %>    </div>    <div class="field">      <%= f.label :body %><br />      <%= f.text\_area :body %>    </div>    <div class="actions">      <%= f.submit %>    </div>  <% end %>    <%= link\_to 'Edit Post', edit\_post\_path(@post) %> |  <%= link\_to 'Back to Posts', posts\_path %> | |

This adds a form on the Post show page that creates a new comment by calling theCommentsController create action. Let’s wire that up:

|  |
| --- |
| class CommentsController < ApplicationController    def create      @post = Post.find(params[:post\_id])      @comment = @post.comments.create(params[:comment])      redirect\_to post\_path(@post)    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, thus the initial call to the find method of the Post model to get the post in question.

In addition, the code takes advantage of some of the methods available for an association. We use the create method on @post.comments to create and save the comment. This will automatically link the comment so that it belongs to that particular post.

Once we have made the new comment, we send the user back to the original post using thepost\_path(@post) helper. As we have already seen, this calls the show action of thePostsController which in turn renders the show.html.erb template. This is where we want the comment to show, so let’s add that to the app/views/posts/show.html.erb.

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>    <h2>Comments</h2>  <% @post.comments.each do |comment| %>    <p>      <b>Commenter:</b>      <%= comment.commenter %>    </p>      <p>      <b>Comment:</b>      <%= comment.body %>    </p>  <% end %>    <h2>Add a comment:</h2>  <%= form\_for([@post, @post.comments.build]) do |f| %>    <div class="field">      <%= f.label :commenter %><br />      <%= f.text\_field :commenter %>    </div>    <div class="field">      <%= f.label :body %><br />      <%= f.text\_area :body %>    </div>    <div class="actions">      <%= f.submit %>    </div>  <% end %>    <br />    <%= link\_to 'Edit Post', edit\_post\_path(@post) %> |  <%= link\_to 'Back to Posts', posts\_path %> | |

Now you can add posts and comments to your blog and have them show up in the right places.

**8 Refactoring**

Now that we have posts and comments working, take a look at theapp/views/posts/show.html.erb template. It is getting long and awkward. We can use partials to clean it up.

**8.1 Rendering Partial Collections**

First we will make a comment partial to extract showing all the comments for the post. Create the file app/views/comments/\_comment.html.erb and put the following into it:

|  |
| --- |
| <p>    <b>Commenter:</b>    <%= comment.commenter %>  </p>    <p>    <b>Comment:</b>    <%= comment.body %>  </p> |

Then you can change app/views/posts/show.html.erb to look like the following:

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>    <h2>Comments</h2>  <%= render @post.comments %>    <h2>Add a comment:</h2>  <%= form\_for([@post, @post.comments.build]) do |f| %>    <div class="field">      <%= f.label :commenter %><br />      <%= f.text\_field :commenter %>    </div>    <div class="field">      <%= f.label :body %><br />      <%= f.text\_area :body %>    </div>    <div class="actions">      <%= f.submit %>    </div>  <% end %>    <br />    <%= link\_to 'Edit Post', edit\_post\_path(@post) %> |  <%= link\_to 'Back to Posts', posts\_path %> | |

This will now render the partial in app/views/comments/\_comment.html.erb once for each comment that is in the @post.comments collection. As the render method iterates over the@post.comments collection, it assigns each comment to a local variable named the same as the partial, in this case comment which is then available in the partial for us to show.

**8.2 Rendering a Partial Form**

Let us also move that new comment section out to its own partial. Again, you create a fileapp/views/comments/\_form.html.erb containing:

|  |
| --- |
| <%= form\_for([@post, @post.comments.build]) do |f| %>    <div class="field">      <%= f.label :commenter %><br />      <%= f.text\_field :commenter %>    </div>    <div class="field">      <%= f.label :body %><br />      <%= f.text\_area :body %>    </div>    <div class="actions">      <%= f.submit %>    </div>  <% end %> |

Then you make the app/views/posts/show.html.erb look like the following:

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>    <h2>Comments</h2>  <%= render @post.comments %>    <h2>Add a comment:</h2>  <%= render "comments/form" %>    <br />    <%= link\_to 'Edit Post', edit\_post\_path(@post) %> |  <%= link\_to 'Back to Posts', posts\_path %> | |

The second render just defines the partial template we want to render, comments/form. Rails is smart enough to spot the forward slash in that string and realize that you want to render the\_form.html.erb file in the app/views/comments directory.

The @post object is available to any partials rendered in the view because we defined it as an instance variable.

**9 Deleting Comments**

Another important feature of a blog is being able to delete spam comments. To do this, we need to implement a link of some sort in the view and a DELETE action in the CommentsController.

So first, let’s add the delete link in the app/views/comments/\_comment.html.erb partial:

|  |
| --- |
| <p>    <b>Commenter:</b>    <%= comment.commenter %>  </p>    <p>    <b>Comment:</b>    <%= comment.body %>  </p>    <p>    <%= link\_to 'Destroy Comment', [comment.post, comment],                 :confirm => 'Are you sure?',                 :method => :delete %>  </p> |

Clicking this new “Destroy Comment” link will fire off a DELETE /posts/:id/comments/:id to ourCommentsController, which can then use this to find the comment we want to delete, so let’s add a destroy action to our controller:

|  |
| --- |
| class CommentsController < ApplicationController      def create      @post = Post.find(params[:post\_id])      @comment = @post.comments.create(params[:comment])      redirect\_to post\_path(@post)    end      def destroy      @post = Post.find(params[:post\_id])      @comment = @post.comments.find(params[:id])      @comment.destroy      redirect\_to post\_path(@post)    end    end |

The destroy action will find the post we are looking at, locate the comment within the@post.comments collection, and then remove it from the database and send us back to the show action for the post.

**9.1 Deleting Associated Objects**

If you delete a post then its associated comments will also need to be deleted. Otherwise they would simply occupy space in the database. Rails allows you to use the dependent option of an association to achieve this. Modify the Post model, app/models/post.rb, as follows:

|  |
| --- |
| class Post < ActiveRecord::Base    attr\_accessible :content, :name, :title      validates :name,  :presence => true    validates :title, :presence => true,                      :length => { :minimum => 5 }    has\_many :comments, :dependent => :destroy  end |

**10 Security**

If you were to publish your blog online, anybody would be able to add, edit and delete posts or delete comments.

Rails provides a very simple HTTP authentication system that will work nicely in this situation.

In the PostsController we need to have a way to block access to the various actions if the person is not authenticated, here we can use the Rails http\_basic\_authenticate\_with method, allowing access to the requested action if that method allows it.

To use the authentication system, we specify it at the top of our PostsController, in this case, we want the user to be authenticated on every action, except for index and show, so we write that:

|  |
| --- |
| class PostsController < ApplicationController      http\_basic\_authenticate\_with :name => "dhh", :password => "secret", :except => [:index, :show]      # GET /posts    # GET /posts.json    def index      @posts = Post.all      respond\_to do |format|  # snipped for brevity |

We also only want to allow authenticated users to delete comments, so in theCommentsController we write:

|  |
| --- |
| class CommentsController < ApplicationController      http\_basic\_authenticate\_with :name => "dhh", :password => "secret", :only => :destroy      def create      @post = Post.find(params[:post\_id])  # snipped for brevity |

Now if you try to create a new post, you will be greeted with a basic HTTP Authentication challenge

**11 Building a Multi-Model Form**

Another feature of your average blog is the ability to tag posts. To implement this feature your application needs to interact with more than one model on a single form. Rails offers support for nested forms.

To demonstrate this, we will 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:

|  |
| --- |
| $ rails generate model tag name:string post:references |

Again, 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 (via theaccepts\_nested\_attributes\_for macro) that you intend to edit tags via posts:

|  |
| --- |
| class Post < ActiveRecord::Base    attr\_accessible :content, :name, :title, :tags\_attributes      validates :name,  :presence => true    validates :title, :presence => true,                      :length => { :minimum => 5 }      has\_many :comments, :dependent => :destroy    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 tells Rails to enable destroying tags through the nested attributes (you’ll handle that by displaying 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.

Also note we had to add :tags\_attributes to the attr\_accessible list. If we didn’t do this there would be a MassAssignmentSecurity exception when we try to update tags through our posts model.

We will modify views/posts/\_form.html.erb to render a partial to make a tag:

|  |
| --- |
| <% @post.tags.build %>  <%= form\_for(@post) do |post\_form| %>    <% if @post.errors.any? %>    <div id="errorExplanation">      <h2><%= pluralize(@post.errors.count, "error") %> prohibited this post from being saved:</h2>      <ul>      <% @post.errors.full\_messages.each do |msg| %>        <li><%= msg %></li>      <% end %>      </ul>    </div>    <% end %>      <div class="field">      <%= post\_form.label :name %><br />      <%= post\_form.text\_field :name %>    </div>    <div class="field">      <%= post\_form.label :title %><br />      <%= post\_form.text\_field :title %>    </div>    <div class="field">      <%= post\_form.label :content %><br />      <%= post\_form.text\_area :content %>    </div>    <h2>Tags</h2>    <%= render :partial => 'tags/form',               :locals => {:form => post\_form} %>    <div class="actions">      <%= post\_form.submit %>    </div>  <% end %> |

Note that we have changed the f in form\_for(@post) do |f| to post\_form to make it easier to understand what is going on.

This example shows another option of the render helper, being able to pass in local variables, in this case, we want the local variable form in the partial to refer to the post\_form object.

We also add a @post.tags.build at the top of this form. This is to make sure there is a new tag ready to have its name filled in by the user. If you do not build the new tag, then the form will not appear as there is no new Tag object ready to create.

Now create the folder app/views/tags and make a file in there called \_form.html.erb which contains the form for the tag:

|  |
| --- |
| <%= form.fields\_for :tags do |tag\_form| %>    <div class="field">      <%= tag\_form.label :name, 'Tag:' %>      <%= tag\_form.text\_field :name %>    </div>    <% unless tag\_form.object.nil? || tag\_form.object.new\_record? %>      <div class="field">        <%= tag\_form.label :\_destroy, 'Remove:' %>        <%= tag\_form.check\_box :\_destroy %>      </div>    <% end %>  <% end %> |

Finally, we will edit the app/views/posts/show.html.erb template to show our tags.

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>    <p>    <b>Tags:</b>    <%= @post.tags.map { |t| t.name }.join(", ") %>  </p>    <h2>Comments</h2>  <%= render @post.comments %>    <h2>Add a comment:</h2>  <%= render "comments/form" %>      <%= link\_to 'Edit Post', edit\_post\_path(@post) %> |  <%= link\_to 'Back to Posts', posts\_path %> | |

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

However, that method call @post.tags.map { |t| t.name }.join(", ") is awkward, we could handle this by making a helper method.

**12 View Helpers**

View Helpers live in app/helpers and provide small snippets of reusable code for views. In our case, we want a method that strings a bunch of objects together using their name attribute and joining them with a comma. As this is for the Post show template, we put it in the PostsHelper.

Open up app/helpers/posts\_helper.rb and add the following:

|  |
| --- |
| module PostsHelper    def join\_tags(post)      post.tags.map { |t| t.name }.join(", ")    end  end |

Now you can edit the view in app/views/posts/show.html.erb to look like this:

|  |
| --- |
| <p id="notice"><%= notice %></p>    <p>    <b>Name:</b>    <%= @post.name %>  </p>    <p>    <b>Title:</b>    <%= @post.title %>  </p>    <p>    <b>Content:</b>    <%= @post.content %>  </p>    <p>    <b>Tags:</b>    <%= join\_tags(@post) %>  </p>    <h2>Comments</h2>  <%= render @post.comments %>    <h2>Add a comment:</h2>  <%= render "comments/form" %>      <%= link\_to 'Edit Post', edit\_post\_path(@post) %> |  <%= link\_to 'Back to Posts', posts\_path %> | |

**13 What’s Next?**

Now that you’ve seen your first Rails application, you should feel free to update it and experiment on your own. But you don’t have to do everything without help. As you need assistance getting up and running with Rails, feel free to consult these support resources:

* The [Ruby on Rails guides](http://guides.rubyonrails.org/v3.2.13/index.html)
* The [Ruby on Rails Tutorial](http://railstutorial.org/book)
* The [Ruby on Rails mailing list](http://groups.google.com/group/rubyonrails-talk)
* The [#rubyonrails](irc://irc.freenode.net/#rubyonrails) channel on irc.freenode.net

Rails also comes with built-in help that you can generate using the rake command-line utility:

* Running rake doc:guides will put a full copy of the Rails Guides in the doc/guidesfolder of your application. Open doc/guides/index.html in your web browser to explore the Guides.
* Running rake doc:rails will put a full copy of the API documentation for Rails in thedoc/api folder of your application. Open doc/api/index.html in your web browser to explore the API documentation.

**14 Configuration Gotchas**

The easiest way to work with Rails is to store all external data as UTF-8. If you don’t, Ruby libraries and Rails will often be able to convert your native data into UTF-8, but this doesn’t always work reliably, so you’re better off ensuring that all external data is UTF-8.

If you have made a mistake in this area, the most common symptom is a black diamond with a question mark inside appearing in the browser. Another common symptom is characters like “Ã¼” appearing instead of “ü”. Rails takes a number of internal steps to mitigate common causes of these problems that can be automatically detected and corrected. However, if you have external data that is not stored as UTF-8, it can occasionally result in these kinds of issues that cannot be automatically detected by Rails and corrected.

Two very common sources of data that are not UTF-8:

* Your text editor: Most text editors (such as Textmate), default to saving files as UTF-8. If your text editor does not, this can result in special characters that you enter in your templates (such as é) to appear as a diamond with a question mark inside in the browser. This also applies to your I18N translation files. Most editors that do not already default toUTF-8 (such as some versions of Dreamweaver) offer a way to change the default to UTF-8. Do so.
* Your database. Rails defaults to converting data from your database into UTF-8 at the boundary. However, if your database is not using UTF-8 internally, it may not be able to store all characters that your users enter. For instance, if your database is using Latin-1 internally, and your user enters a Russian, Hebrew, or Japanese character, the data will be lost forever once it enters the database. If possible, use UTF-8 as the internal storage of your database.

**Feedback**

You're encouraged to help improve the quality of this guide.

If you see any typos or factual errors you are confident to patch, please clone [docrails](https://github.com/lifo/docrails) and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. [docrails](https://github.com/lifo/docrails) is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Check the [Ruby on Rails Guides Guidelines](http://guides.rubyonrails.org/v3.2.13/ruby_on_rails_guides_guidelines.html) for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please [open an issue](https://github.com/rails/rails/issues).

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the [rubyonrails-docs mailing list](http://groups.google.com/group/rubyonrails-docs).

This work is licensed under a [Creative Commons Attribution-Share Alike 3.0](http://creativecommons.org/licenses/by-sa/3.0/) License

"Rails", "Ruby on Rails", and the Rails logo are trademarks of David Heinemeier Hansson. All rights reserved.