Preface

This book is a LaTeXport of Ruby on Rails' documentation available at http://guides.rubyonrails.org, so all credit should go to those guys.

It was extracted from the page itself using **html2latex** and then bulk edited with **vim** and **LaTeXila** (on Debian *Wheezy*).

As of September 2012, the book spans the basic chapters from *Getting Started...* to *Rails Routing...*, through the **MVC** architecture. They were treated (& read) one-at-a-time so their editions are slighty different and tend toward better style as chapter numbers grow – see code display, for example. The idea is to standarize it to best practices whenever time allows it.

It is a `pocket` book. It's dimensions are a quarter of Chilean Oficio (216x279mm, a bit shorter than U.S. Office, if not deceiving myself). This is because it's the cheapest paper around, that's all. Anyone willing to take this to an A6 or whatever harmonic ratio of $\sqrt{2}, pleasedofork!I'meagertoseehowitlooks.Beawarethattableswillgivewor$

Chapter 1

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 3.2. Some of the code shown here will not work in earlier versions of Rails.

1.1 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 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 packaging system
  - If you want to learn more about RubyGems, please read the RubyGems User Guide
- A working installation of the SQLite3 Database

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
- Programming Ruby
- Why's (Poignant) Guide to Ruby

Also, the example code for this guide is available in the rails github:

https://github.com/rails/rails repository in rails/railties/guides/code/g

1.2 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.

1.2.1 The MVC Architecture

At the core of Rails is 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, 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.

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.

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.

1.2.2 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

Action Pack

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

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. View templates are covered in more detail in another guide called Layouts and Rendering.

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.

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.

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 ORM frameworks in place of Active Record if your application needs this.

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.

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 with CRUD semantics.

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.

Railties

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

1.2.3 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. 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 by Stefan Tilkov
- An Introduction to REST (video tutorial) by Joe Gregorio
- Representational State Transfer article in Wikipedia
- How to GET a Cup of Coffee by Jim Webber, Savas Parastatidis & Ian Robinson

1.3 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.

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>

1.3.1 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.

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.

1.3.2 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 running

```
$ rails 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 folder for the remainder of this guide.
config/	Configure your application's runtime rules, routes, data detail in Configuring Rails Applications
config.ru	Rack configuration for Rack based servers used to start
db/	Contains your current database schema, as well as the
doc/	In-depth documentation for your application.
Gemfile	
Gemfile.lock	These files allow you to specify what gem dependencies
lib/	Extended modules for your application.
log/	Application log files.
public/	The only folder seen to the world as-is. Contains the st
Rakefile	This file locates and loads tasks that can be run from t are defined throughout the components of Rails. Ratl add your own tasks by adding files to the lib/tasks dire
README.rdoc	This is a brief instruction manual for your application. what your application does, how to set it up, and so or
script/	Contains the rails script that starts your app and can or run your application.
test/	Unit tests, fixtures, and other test apparatus. These ar
tmp/	Temporary files
vendor/	A place for all third-party code. In a typical Rails app Rails source code (if you optionally install it into your pr prepackaged functionality.

1.3.3 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.

Configuring an SQLite3 Database

Rails comes with built-in support for SQLite3, 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.

Configuring a MySQL Database

If you choose to use MySQL instead of the shipped SQLite3 database, your `config/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 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:	

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

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:	

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.

1.3.4 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`.

1.4 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.

1.4.1 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.

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:3`
You should see Rails' default information page:

../rails_welcome.png

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.

1.4.2 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.ht` 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>
```

1.4.3 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/`, 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 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. 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 in your browser, you'll see `Hello, Rails!`.

For more information about routing, refer to Rails Routing from the Outside In.

1.5 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.

1.6 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:te
```

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

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration name will
app/models/post.rb	The Post
test/unit/post_test.rb	Unit testi
test/fixtures/posts.yml	Sample po
config/routes.rb	Edited to
app/controllers/posts_controller.rb	The Posts
app/views/posts/index.html.erb	A view to
app/views/posts/edit.html.erb	A view to
app/views/posts/show.html.erb	A view to
app/views/posts/new.html.erb	A view to
app/views/posts/_form.html.erb	A partial used in ed
test/functional/posts_controller_test.rb	Functiona
app/helpers/posts_helper.rb	Helper fu
test/unit/helpers/posts_helper_test.rb	Unit testi
app/assets/javascripts/posts.js.coffee	CoffeeScri
app/assets/stylesheets/posts.css.scss	Cascading
app/assets/stylesheets/scaffolds.css.scss	Cascading 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 guide.

1.6.1 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 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 your `config/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`.

1.6.2 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.

1.6.3 Working with Posts in the Browser

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

../posts_index.png

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.

1.6.4 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).

1.6.5 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
  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

1.6.6 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 s
```

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.

1.6.7 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_con` 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.

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's `app/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 HTML would 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.

1.6.8 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, like `app/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 in `app/views/layouts/applic`
Open this layout in your editor and modify the `body` 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.

1.6.9 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.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 of HTML 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 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 HTML output.

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 HTTPPOST 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, `create` returns 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 `Post` as 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.”

1.6.10 Showing an Individual Post

When you click the `show` link for a post on the index page, it will bring you to a URL 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.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 %>
```

1.6.11 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>

<%= 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 `post` parameter (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.

1.6.12 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.

1.7 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.

1.7.1 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:ref
```

This command will generate four files:

File	Purpos
db/migrate/20100207235629_create_comments.rb	Migratio database tamp)
app/models/comment.rb	The Con
test/unit/comment_test.rb	Unit tes
test/fixtures/comments.yml	Sample

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 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) ============================
```

1.7.2 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 guide.

1.7.3 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 guide.

1.7.4 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
test/functional/comments_controller_test.rb	The functional tests for th
app/helpers/comments_helper.rb	A view helper file
test/unit/helpers/comments_helper_test.rb	The unit tests for the help
app/assets/javascripts/comment.js.coffee	CoffeeScript for the contro
app/assets/stylesheets/comment.css.scss	Cascading style sheet for t

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.h` 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 the `CommentsControllercreate` 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 the `post_path(@post)` helper. As we have already seen, this calls the `show` action of the `PostsController` 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.e`

```
<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 %>
```

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

1.8 Refactoring

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

1.8.1 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.` 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.

1.8.2 Rendering a Partial Form

Let us also move that new comment section out to its own partial. Again, you create a file `app/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.

1.9 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.ht` 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 our `CommentsController`, 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.

1.9.1 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
```

1.10 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 => "secre

  # 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 the CommentsController we write:

```
class CommentsController < ApplicationController

  http_basic_authenticate_with :name => "dhh", :password => "s

  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

`../challenge.png`

1.11 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 the `accepts_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") %> prohibit
    <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.

1.12 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 %> |
```

1.13 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
- The Ruby on Rails Tutorial
- The Ruby on Rails mailing list
- The #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/guides` folder 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 the `doc/api` folder of your application. Open `doc/api/index.html` in your web browser to explore the API documentation.

1.14 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 "Ã" 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 to UTF-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.

Chapter 2

Migrations

Migrations are a convenient way for you to alter your database in a structured and organized manner. You could edit fragments of SQL by hand but you would then be responsible for telling other developers that they need to go and run them. You'd also have to keep track of which changes need to be run against the production machines next time you deploy.

Active Record tracks which migrations have already been run so all you have to do is update your source and run `rake db:migrate`. Active Record will work out which migrations should be run. It will also update your `db/schema.rb` file to match the structure of your database.

Migrations also allow you to describe these transformations using Ruby. The great thing about this is that (like most of Active Record's functionality) it is database independent: you don't need to worry about the precise syntax of `CREATE TABLE` any more than you worry about variations on `SELECT *` (you can drop down to raw SQL for database specific features). For example you could use SQLite3 in development, but MySQL in production.

In this guide, you'll learn all about migrations including:

- The generators you can use to create them
- The methods Active Record provides to manipulate your database
- The Rake tasks that manipulate them

- How they relate to `schema.rb`

2.1 Anatomy of a Migration

Before we dive into the details of a migration, here are a few examples of the sorts of things you can do:

```
class CreateProducts < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end

  def down
    drop_table :products
  end
end
```

This migration adds a table called `products` with a string column called `name` and a text column called `description`. A primary key column called `id` will also be added, however since this is the default we do not need to ask for this. The timestamp columns `created_at` and `updated_at` which Active Record populates automatically will also be added. Reversing this migration is as simple as dropping the table.

Migrations are not limited to changing the schema. You can also use them to fix bad data in the database or populate new fields:

```
class AddReceiveNewsletterToUsers < ActiveRecord::Migration
  def up
    change_table :users do |t|
      t.boolean :receive_newsletter, :default => false
    end
    User.update_all ["receive_newsletter = ?", true]
  end

  def down
    remove_column :users, :receive_newsletter
  end
end
```

Some caveats apply to using models in your migrations.

This migration adds a `receive_newsletter` column to the `users` table. We want it to default to `false` for new users, but existing users are considered to have already opted in, so we use the User model to set the flag to `true` for existing users.

Rails 3.1 makes migrations smarter by providing a new `change` method. This method is preferred for writing constructive migrations (adding columns or tables). The migration knows how to migrate your database and reverse it when the migration is rolled back without the need to write a separate `down` method.

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

2.1.1 Migrations are Classes

A migration is a subclass of `ActiveRecord::Migration` that implements two methods: `up` (perform the required transformations) and `down` (revert them).

Active Record provides methods that perform common data definition tasks in a database independent way (you'll read about them in detail later):

- `add_column`
- `add_index`
- `change_column`
- `change_table`
- `create_table`
- `drop_table`
- `remove_column`
- `remove_index`
- `rename_column`

If you need to perform tasks specific to your database (for example create a foreign key constraint) then the `execute` method allows you to execute arbitrary SQL. A migration is just a regular Ruby class so you're not limited to these functions. For example after adding a column you could write code to set the value of that column for existing records (if necessary using your models).

On databases that support transactions with statements that change the schema (such as PostgreSQL or SQLite3), migrations are wrapped in a transaction. If the database does not support this (for example MySQL) then when a migration fails the parts of it that succeeded will not be rolled back. You will have to rollback the changes that were made by hand.

2.1.2 What's in a Name

Migrations are stored as files in the `db/migrate` directory, one for each migration class. The name of the file is of the form `YYYYMMDDHHMMSS_crea` that is to say a UTC timestamp identifying the migration followed by an underscore followed by the name of the migration. The name of the migration class (CamelCased version) should match the latter part of the file name. For example `20080906120000_create_products.rb` should define class `CreateProducts` and `20080906120001_add_details_to_pro` should define `AddDetailsToProducts`. If you do feel the need to change the file name then you have to update the name of the class inside or Rails will complain about a missing class.

Internally Rails only uses the migration's number (the timestamp) to identify them. Prior to Rails 2.1 the migration number started at 1 and was incremented each time a migration was generated. With multiple developers it was easy for these to clash requiring you to rollback migrations and renumber them. With Rails 2.1+ this is largely avoided by using the creation time of the migration to identify them. You can revert to the old numbering scheme by adding the following line to `config/application.rb`.

```
config.active_record.timestamped_migrations = false
```

The combination of timestamps and recording which migrations have been run allows Rails to handle common situations that occur with multiple developers.

For example Alice adds migrations `20080906120000` and `20080906123000` and Bob adds `20080906124500` and runs it. Alice finishes her changes and checks in her migrations and Bob pulls down the latest changes. When Bob runs `rake db:migrate`, Rails knows that it has not run Alice's two migrations so it executes the `up` method for each migration.

Of course this is no substitution for communication within the team. For example, if Alice's migration removed a table that Bob's migration assumed to exist, then trouble would certainly strike.

2.1.3 Changing Migrations

Occasionally you will make a mistake when writing a migration. If you have already run the migration then you cannot just edit the migration and run the migration again: Rails thinks it has already run the migration and so will do nothing when you run `rake db:migrate`. You must rollback the migration (for example with `rake db:rollback`), edit your migration and then run `rake db:migrate` to run the corrected version.

In general editing existing migrations is not a good idea: you will be creating extra work for yourself and your co-workers and cause major headaches if the existing version of the migration has already been run on production machines. Instead, you should write a new migration that performs the changes you require. Editing a freshly generated migration that has not yet been committed to source control (or, more generally, which has not been propagated beyond your development machine) is relatively harmless.

2.1.4 Supported Types

Active Record supports the following database column types:

- `:binary`
- `:boolean`
- `:date`
- `:datetime`
- `:decimal`
- `:float`
- `:integer`
- `:primary_key`
- `:string`

- `:text`
- `:time`
- `:timestamp`

These will be mapped onto an appropriate underlying database type. For example, with MySQL the type `:string` is mapped to `VARCHAR(255)`. You can create columns of types not supported by Active Record when using the non-sexy syntax, for example

```
create_table :products do |t|
  t.column :name, 'polygon', :null => false
end
```

This may however hinder portability to other databases.

2.2 Creating a Migration

2.2.1 Creating a Model

The model and scaffold generators will create migrations appropriate for adding a new model. This migration will already contain instructions for creating the relevant table. If you tell Rails what columns you want, then statements for adding these columns will also be created. For example, running

```
$ rails generate model Product name:string description:text
```

will create a migration that looks like this

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

You can append as many column name/type pairs as you want. By default, the generated migration will include `t.timestamps` (which creates the `updated_at` and `created_at` columns that are automatically populated by Active Record).

2.2.2 Creating a Standalone Migration

If you are creating migrations for other purposes (for example to add a column to an existing table) then you can also use the migration generator:

```
$ rails generate migration AddPartNumberToProducts
```

This will create an empty but appropriately named migration:

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
  end
end
```

If the migration name is of the form "AddXXXToYYY" or "RemoveXXXFromYYY" and is followed by a list of column names and types then a migration containing the appropriate `add_column` and `remove_column` statements will be created.

```
$ rails generate migration AddPartNumberToProducts part_number:str
```

will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```

Similarly,

```
$ rails generate migration RemovePartNumberFromProducts part_numbe
```

generates

```
class RemovePartNumberFromProducts < ActiveRecord::Migration
  def up
    remove_column :products, :part_number
  end

  def down
    add_column :products, :part_number, :string
  end
end
```

You are not limited to one magically generated column, for example

```
$ rails generate migration AddDetailsToProducts part_number:string
```

generates

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

As always, what has been generated for you is just a starting point. You can add or remove from it as you see fit by editing the db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb file.

The generated migration file for destructive migrations will still be old-style using the `up` and `down` methods. This is because Rails needs to know the original data types defined when you made the original changes.

2.3 Writing a Migration

Once you have created your migration using one of the generators it's time to get to work!

2.3.1 Creating a Table

Migration method `create_table` will be one of your workhorses. A typical use would be

```
create_table :products do |t|
  t.string :name
end
```

which creates a `products` table with a column called `name` (and as discussed below, an implicit `id` column).

The object yielded to the block allows you to create columns on the table. There are two ways of doing it. The first (traditional) form looks like

```
create_table :products do |t|
  t.column :name, :string, :null => false
```

```
end
```

The second form, the so called "sexy" migration, drops the somewhat redundant `column` method. Instead, the `string`, `integer`, etc. methods create a column of that type. Subsequent parameters are the same.

```
create_table :products do |t|
  t.string :name, :null => false
end
```

By default, `create_table` will create a primary key called `id`. You can change the name of the primary key with the `:primary_key` option (don't forget to update the corresponding model) or, if you don't want a primary key at all (for example for a HABTM join table), you can pass the option `:id => false`. If you need to pass database specific options you can place an SQL fragment in the `:options` option. For example,

```
create_table :products, :options => "ENGINE=BLACKHOLE" do |t|
  t.string :name, :null => false
end
```

will append `ENGINE=BLACKHOLE` to the SQL statement used to create the table (when using MySQL, the default is `ENGINE=InnoDB`).

2.3.2 Changing Tables

A close cousin of `create_table` is `change_table`, used for changing existing tables. It is used in a similar fashion to `create_table` but the object yielded to the block knows more tricks. For example

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

removes the `description` and `name` columns, creates a `part_number` string column and adds an index on it. Finally it renames the `upccode` column.

2.3.3 Special Helpers

Active Record provides some shortcuts for common functionality. It is for example very common to add both the `created_at` and `updated_at` columns and so there is a method that does exactly that:

```
create_table :products do |t|
  t.timestamps
end
```

will create a new products table with those two columns (plus the `id` column) whereas

```
change_table :products do |t|
  t.timestamps
end
```

adds those columns to an existing table.

Another helper is called `references` (also available as `belongs_to`). In its simplest form it just adds some readability.

```
create_table :products do |t|
  t.references :category
end
```

will create a `category_id` column of the appropriate type. Note that you pass the model name, not the column name. Active Record adds the `_id` for you. If you have polymorphic `belongs_to` associations then `references` will add both of the columns required:

```
create_table :products do |t|
  t.references :attachment, :polymorphic => {:default => 'Photo'}
end
```

will add an `attachment_id` column and a string `attachment_type` column with a default value of 'Photo'.

The `references` helper does not actually create foreign key constraints for you. You will need to use `execute` or a plugin that adds foreign key support.

If the helpers provided by Active Record aren't enough you can use the `execute` method to execute arbitrary SQL.

For more details and examples of individual methods, check the API documentation, in particular the documentation for `ActiveRecord::Con` (which provides the methods available in the `up` and `down` methods), `ActiveRecord::ConnectionAdapters::TableDefinition` (which provides the methods available on the object yielded by `create_table`) and `ActiveRecord::ConnectionAdapters::Table` (which provides the methods available on the object yielded by `change_table`).

2.3.4 Using the `change` Method

The `change` method removes the need to write both `up` and `down` methods in those cases that Rails know how to revert the changes automatically. Currently, the `change` method supports only these migration definitions:

- `add_column`
- `add_index`
- `add_timestamps`

- `create_table`
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `rename_table`

If you're going to need to use any other methods, you'll have to write the `up` and `down` methods instead of using the `change` method.

2.3.5 Using the `up`/`down` Methods

The `down` method of your migration should revert the transformations done by the `up` method. In other words, the database schema should be unchanged if you do an `up` followed by a `down`. For example, if you create a table in the `up` method, you should drop it in the `down` method. It is wise to reverse the transformations in precisely the reverse order they were made in the `up` method. For example,

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.references :category
    end
    #add a foreign key
    execute <<-SQL
      ALTER TABLE products
        ADD CONSTRAINT fk_products_categories
        FOREIGN KEY (category_id)
        REFERENCES categories(id)
    SQL
    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url
    execute <<-SQL
      ALTER TABLE products
        DROP FOREIGN KEY fk_products_categories
    SQL
    drop_table :products
  end
end
```

Sometimes your migration will do something which is just plain irreversible; for example, it might destroy some data. In such cases, you can raise `ActiveRecord::IrreversibleMigration` from your `down` method. If someone tries to revert your migration, an error message will be displayed saying that it can't be done.

2.4 Running Migrations

Rails provides a set of rake tasks to work with migrations which boil down to running certain sets of migrations.

The very first migration related rake task you will use will probably be `rake db:migrate`. In its most basic form it just runs the `up` or `change` method for all the migrations that have not yet been run. If there are no such migrations, it exits. It will run these migrations in order based on the date of the migration.

Note that running the `db:migrate` also invokes the `db:schema:dump` task, which will update your db/schema.rb file to match the structure of your database.

If you specify a target version, Active Record will run the required migrations (up, down or change) until it has reached the specified version. The version is the numerical prefix on the migration's filename. For example, to migrate to version 20080906120000 run

```
$ rake db:migrate VERSION=20080906120000
```

If version 20080906120000 is greater than the current version (i.e., it is migrating upwards), this will run the `up` method on all migrations up to and including 20080906120000, and will not execute any later migrations. If migrating downwards, this will run the `down` method on all the migrations down to, but not including, 20080906120000.

2.4.1 Rolling Back

A common task is to rollback the last migration, for example if you made a mistake in it and wish to correct it. Rather than tracking down the version number associated with the previous migration you can run

```
$ rake db:rollback
```

This will run the `down` method from the latest migration. If you need to undo several migrations you can provide a `STEP` parameter:

```
$ rake db:rollback STEP=3
```

will run the `down` method from the last 3 migrations.

The `db:migrate:redo` task is a shortcut for doing a rollback and then migrating back up again. As with the `db:rollback` task, you can use the `STEP` parameter if you need to go more than one version back, for example

```
$ rake db:migrate:redo STEP=3
```

Neither of these Rake tasks do anything you could not do with `db:migrate`. They are simply more convenient, since you do not need to explicitly specify the version to migrate to.

2.4.2 Resetting the database

The `rake db:reset` task will drop the database, recreate it and load the current schema into it.

This is not the same as running all the migrations – see the section on schema.rb.

2.4.3 Running specific migrations

If you need to run a specific migration up or down, the `db:migrate:up` and `db:migrate:down` tasks will do that. Just specify the appropriate version and the corresponding migration will have its `up` or `down` method invoked, for example,

```
$ rake db:migrate:up VERSION=20080906120000
```

will run the `up` method from the 20080906120000 migration. These tasks still check whether the migration has already run, so for example **db:migrate:up VERSION=20080906120000** will do nothing if Active Record believes that 20080906120000 has already been run.

2.4.4 Changing the output of running migrations

By default migrations tell you exactly what they're doing and how long it took. A migration creating a table and adding an index might produce output like this

```
==  CreateProducts: migrating ==================================
-- create_table(:products)
   -> 0.0028s
==  CreateProducts: migrated (0.0028s) =========================
```

Several methods are provided in migrations that allow you to control all this:

Method	Purpose
suppress_messages	Takes a block as an argument and suppresses any output generated by the block.
say	Takes a message argument and outputs it as is. A second boolean argument can be passed to specify whether to indent or not.
say_with_time	Outputs text along with how long it took to run its block. If the block returns an integer it assumes it is the number of rows affected.

For example, this migration

```
class CreateProducts < ActiveRecord::Migration
  def change
    suppress_messages do
      create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
      end
    end
    say "Created a table"
    suppress_messages {add_index :products, :name}
    say "and an index!", true
    say_with_time 'Waiting for a while' do
      sleep 10
      250
    end
  end
end
```

generates the following output

```
==  CreateProducts: migrating ======================================
-- Created a table
   -> and an index!
-- Waiting for a while
   -> 10.0013s
   -> 250 rows
==  CreateProducts: migrated (10.0054s) ============================
```

If you want Active Record to not output anything, then running

`rake db:migrate VERBOSE=false} will suppress all output.`

2.5 Using Models in Your Migrations

When creating or updating data in a migration it is often tempting to use one of your models. After all, they exist to provide easy access to the underlying data. This can be done, but some caution should be observed.

For example, problems occur when the model uses database columns which are (1) not currently in the database and (2) will be created by this or a subsequent migration.

Consider this example, where Alice and Bob are working on the same code base which contains a `Product` model:

Bob goes on vacation.

Alice creates a migration for the `products` table which adds a new column and initializes it. She also adds a validation to the `Product` model for the new column.

```
# db/migrate/20100513121110_add_flag_to_product.rb

class AddFlagToProduct < ActiveRecord::Migration
  def change
    add_column :products, :flag, :boolean
    Product.all.each do |product|
      product.update_attributes!(:flag => 'false')
    end
  end
end
```

```
# app/model/product.rb

class Product < ActiveRecord::Base
  validates :flag, :presence => true
end
```

Alice adds a second migration which adds and initializes another column to the `products` table and also adds a validation to the `Product` model for the new column.

```
# db/migrate/20100515121110_add_fuzz_to_product.rb

class AddFuzzToProduct < ActiveRecord::Migration
  def change
    add_column :products, :fuzz, :string
    Product.all.each do |product|
      product.update_attributes! :fuzz => 'fuzzy'
    end
  end
end

# app/model/product.rb

class Product < ActiveRecord::Base
  validates :flag, :fuzz, :presence => true
end
```

Both migrations work for Alice.

Bob comes back from vacation and:

1. Updates the source – which contains both migrations and the latests version of the Product model.
2. Runs outstanding migrations with `rake db:migrate`, which includes the one that updates the `Product` model.

The migration crashes because when the model attempts to save, it tries to validate the second added column, which is not in the database when the first migration runs:

```
rake aborted!
An error has occurred, this and all later migrations canceled:

undefined method 'fuzz' for #<Product:0x000001049b14a0>
```

A fix for this is to create a local model within the migration. This keeps rails from running the validations, so that the migrations run to completion.

When using a faux model, it's a good idea to call `Product.reset_column_` to refresh the `ActiveRecord` cache for the `Product` model prior to updating data in the database.

If Alice had done this instead, there would have been no problem:

```
# db/migrate/20100513121110_add_flag_to_product.rb

class AddFlagToProduct < ActiveRecord::Migration
  class Product < ActiveRecord::Base
  end

  def change
    add_column :products, :flag, :integer
    Product.reset_column_information
    Product.all.each do |product|
      product.update_attributes!(:flag => false)
    end
  end
end
```

```
# db/migrate/20100515121110_add_fuzz_to_product.rb

class AddFuzzToProduct < ActiveRecord::Migration
  class Product < ActiveRecord::Base
  end

  def change
    add_column :products, :fuzz, :string
    Product.reset_column_information
    Product.all.each do |product|
      product.update_attributes!(:fuzz => 'fuzzy')
    end
  end
end
```

2.6 Schema Dumping and You

2.6.1 What are Schema Files for?

Migrations, mighty as they may be, are not the authoritative source for your database schema. That role falls to either `db/schema.rb` or an SQL file which Active Record generates by examining the database. They are not designed to be edited, they just represent the current state of the database.

There is no need (and it is error prone) to deploy a new instance of an app by replaying the entire migration history. It is much simpler and faster to just load into the database a description of the current schema.

For example, this is how the test database is created: the current development database is dumped (either to `db/schema.rb` or `db/structure.sql`) and then loaded into the test database.

Schema files are also useful if you want a quick look at what attributes an Active Record object has. This information is not in the model's code and is frequently spread across several migrations, but the information is nicely summed up in the schema file. The annotate_models gem automatically adds and updates comments at the top of each model summarizing the schema if you desire that functionality.

2.6.2 Types of Schema Dumps

There are two ways to dump the schema. This is set in `config/application.` by the `config.active_record.schema_format` setting, which may be either `:sql` or `:ruby`.

If `:ruby` is selected then the schema is stored in `db/schema.rb`. If you look at this file you'll find that it looks an awful lot like one very big migration:

```
ActiveRecord::Schema.define(:version => 20080906171750) do
  create_table "authors", :force => true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", :force => true do |t|
    t.string   "name"
    t.text "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string "part_number"
  end
end
```

In many ways this is exactly what it is. This file is created by inspecting the database and expressing its structure using `create_table`, `add_index`, and so on. Because this is database-independent, it could be loaded into any database that Active Record supports. This could be very useful if you were to distribute an application that is able to run against multiple databases.

There is however a trade-off: `db/schema.rb` cannot express database specific items such as foreign key constraints, triggers, or stored procedures. While in a migration you can execute custom SQL statements, the schema dumper cannot reconstitute those statements from the database. If you are using features like this, then you should set the schema format to `:sql`.

Instead of using Active Record's schema dumper, the database's structure will be dumped using a tool specific to the database (via the `db:structure:dump` Rake task) into `db/structure.sql`. For example, for the PostgreSQL RDBMS, the `pg_dump` utility is used. For MySQL, this file will contain the output of **SHOWCREATETABLE** for the various tables. Loading these schemas is simply a question of executing the SQL statements they contain. By definition, this will create a perfect copy of the database's structure. Using the `:sql` schema format will, however, prevent loading the schema into a RDBMS other than the one used to create it.

2.6.3 Schema Dumps and Source Control

Because schema dumps are the authoritative source for your database schema, it is strongly recommended that you check them into source control.

2.7 Active Record and Referential Integrity

The Active Record way claims that intelligence belongs in your models, not in the database. As such, features such as triggers or foreign key constraints, which push some of that intelligence back into the database, are not heavily used.

Validations such as `validates :foreign_key, :uniqueness => true` are one way in which models can enforce data integrity. The `:dependent` option on associations allows models to automatically destroy child objects when the parent is destroyed. Like anything which operates at the application level, these cannot guarantee referential integrity and so some people augment them with foreign key constraints in the database.

Although Active Record does not provide any tools for working directly with such features, the `execute` method can be used to execute arbitrary SQL. You could also use some plugin like foreigner which add foreign key support to Active Record (including support for dumping foreign keys in `db/schema.rb`).

Chapter 3

Active Record Validations and Callbacks

This guide teaches you how to hook into the life cycle of your Active Record objects. You will learn how to validate the state of objects before they go into the database, and how to perform custom operations at certain points in the object life cycle.

After reading this guide and trying out the presented concepts, we hope that you'll be able to:

- Understand the life cycle of Active Record objects
- Use the built-in Active Record validation helpers
- Create your own custom validation methods
- Work with the error messages generated by the validation process
- Create callback methods that respond to events in the object life cycle
- Create special classes that encapsulate common behavior for your callbacks

- Create Observers that respond to life cycle events outside of the original class

3.1 The Object Life Cycle

During the normal operation of a Rails application, objects may be created, updated, and destroyed. Active Record provides hooks into this object life cycle so that you can control your application and its data.

Validations allow you to ensure that only valid data is stored in your database. Callbacks and observers allow you to trigger logic before or after an alteration of an object's state.

3.2 Validations Overview

Before you dive into the detail of validations in Rails, you should understand a bit about how validations fit into the big picture.

3.2.1 Why Use Validations?

Validations are used to ensure that only valid data is saved into your database. For example, it may be important to your application to ensure that every user provides a valid email address and mailing address.

There are several ways to validate data before it is saved into your database, including native database constraints, client-side validations, controller-level validations, and model-level validations:

- Database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult. However, if your database is used by other applications, it may be a good idea to use some constraints at the database level. Additionally, database-level validations can safely handle some things (such as uniqueness in heavily-used tables) that can be difficult to implement otherwise.

- Client-side validations can be useful, but are generally unreliable if used alone. If they are implemented using JavaScript, they may be bypassed if JavaScript is turned off in the user's browser. However, if combined with other techniques, client-side validation can be a convenient way to provide users with immediate feedback as they use your site.
- Controller-level validations can be tempting to use, but often become unwieldy and difficult to test and maintain. Whenever possible, it's a good idea to keep your controllers skinny, as it will make your application a pleasure to work with in the long run.
- Model-level validations are the best way to ensure that only valid data is saved into your database. They are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain. Rails makes them easy to use, provides built-in helpers for common needs, and allows you to create your own validation methods as well.

3.2.2 When Does Validation Happen?

There are two kinds of Active Record objects: those that correspond to a row inside your database and those that do not. When you create a fresh object, for example using the `new` method, that object does not belong to the database yet. Once you call `save` upon that object it will be saved into the appropriate database table. Active Record uses the `new_record?` instance method to determine whether an object is already in the database or not. Consider the following simple Active Record class:

```
class Person < ActiveRecord::Base
end
```

We can see how it works by looking at some `rails console` output:

```
>> p = Person.new(:name => "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, :updated_at: nil>
>> p.new_record?
=> true
>> p.save
=> true
>> p.new_record?
=> false
```

Creating and saving a new record will send an SQL`INSERT` operation to the database. Updating an existing record will send an SQL`UPDATE` operation instead. Validations are typically run before these commands are sent to the database. If any validations fail, the object will be marked as invalid and Active Record will not perform the `INSERT` or `UPDATE` operation. This helps to avoid storing an invalid object in the database. You can choose to have specific validations run when an object is created, saved, or updated.

There are many ways to change the state of an object in the database. Some methods will trigger validations, but some will not. This means that it's possible to save an object in the database in an invalid state if you aren't careful.

The following methods trigger validations, and will save the object to the database only if the object is valid:

- `create`
- `create!`
- `save`
- `save!`
- `update`

- `update_attributes`
- `update_attributes!`

The bang versions (e.g. `save!`) raise an exception if the record is invalid. The non-bang versions don't: `save` and `update_attributes` return `false`, `create` and `update` just return the objects.

3.2.3 Skipping Validations

The following methods skip validations, and will save the object to the database regardless of its validity. They should be used with caution.

- `decrement!`
- `decrement_counter`
- `increment!`
- `increment_counter`
- `toggle!`
- `touch`
- `update_all`
- `update_attribute`
- `update_column`
- `update_counters`

Note that `save` also has the ability to skip validations if passed `:validate => false` as argument. This technique should be used with caution.

- `save(:validate => false)`

3.2.4 `valid?` and `invalid?`

To verify whether or not an object is valid, Rails uses the `valid?` method. You can also use this method on your own. `valid?` triggers your validations and returns true if no errors were found in the object, and false otherwise.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

Person.create(:name => "John Doe").valid? # => true
Person.create(:name => nil).valid? # => false
```

After Active Record has performed validations, any errors found can be accessed through the `errors` instance method, which returns a collection of errors. By definition, an object is valid if this collection is empty after running validations.

Note that an object instantiated with `new` will not report errors even if it's technically invalid, because validations are not run when using `new`.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

>> p = Person.new
=> #<Person id: nil, name: nil>
>> p.errors
=> {}

>> p.valid?
=> false
>> p.errors
=> {:name=>["can’t be blank"]}

>> p = Person.create
=> #<Person id: nil, name: nil>
>> p.errors
=> {:name=>["can’t be blank"]}

>> p.save
=> false

>> p.save!
=> ActiveRecord::RecordInvalid: Validation failed: Name can’t be blank

>> Person.create!
=> ActiveRecord::RecordInvalid: Validation failed: Name can’t be blank
```

`invalid?` is simply the inverse of `valid?`. `invalid?` triggers your validations, returning true if any errors were found in the object, and false otherwise.

3.2.5 `errors[]`

To verify whether or not a particular attribute of an object is valid, you can use `errors[:attribute]`. It returns an array of all the errors for `:attribute`. If there are no errors on the specified attribute, an empty array is returned.

This method is only useful after validations have been run, because it only inspects the errors collection and does not trigger validations itself. It's different from the `ActiveRecord::Base#invalid?` method explained above because it doesn't verify the validity of the object as a whole. It only checks to see whether there are errors found on an individual attribute of the object.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

We'll cover validation errors in greater depth in the Working with Validation Errors section. For now, let's turn to the built-in validation helpers that Rails provides by default.

3.3 Validation Helpers

Active Record offers many pre-defined validation helpers that you can use directly inside your class definitions. These helpers provide common validation rules. Every time a validation fails, an error message is added to the object's `errors` collection, and this message is associated with the attribute being validated.

Each helper accepts an arbitrary number of attribute names, so with a single line of code you can add the same kind of validation to several attributes.

All of them accept the `:on` and `:message` options, which define when the validation should be run and what message should be added to the `errors` collection if it fails, respectively. The `:on` option takes one of the values `:save` (the default), `:create` or `:update`. There is a default error message for each one of the validation helpers. These messages are used when the `:message` option isn't specified. Let's take a look at each one of the available helpers.

3.3.1 `acceptance`

Validates that a checkbox on the user interface was checked when a form was submitted. This is typically used when the user needs to agree to your application's terms of service, confirm reading some text, or any similar concept. This validation is very specific to web applications and this 'acceptance' does not need to be recorded anywhere in your database (if you don't have a field for it, the helper will just create a virtual attribute).

```
class Person < ActiveRecord::Base
  validates :terms_of_service, :acceptance => true
end
```

The default error message for this helper is "must be accepted".

It can receive an `:accept` option, which determines the value that will be considered acceptance. It defaults to "1" and can be easily changed.

```
class Person < ActiveRecord::Base
  validates :terms_of_service, :acceptance => { :accept => 'yes' }
end
```

3.3.2 `validates_associated`

You should use this helper when your model has associations with other models and they also need to be validated. When you try to save your object, `valid?` will be called upon each one of the associated objects.

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```

This validation will work with all of the association types.

Don't use `validates_associated` on both ends of your associations. They would call each other in an infinite loop.

The default error message for `validates_associated` is "is invalid". Note that each associated object will contain its own `errors` collection; errors do not bubble up to the calling model.

3.3.3 `confirmation`

You should use this helper when you have two text fields that should receive exactly the same content. For example, you may want to confirm an email address or a password. This validation creates a virtual attribute whose name is the name of the field that has to be confirmed with "_confirmation" appended.

```
class Person < ActiveRecord::Base
  validates :email, :confirmation => true
end
```

In your view template you could use something like

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

This check is performed only if `email_confirmation` is not `nil`. To require confirmation, make sure to add a presence check for the confirmation attribute (we'll take a look at `presence` later on this guide):

```
class Person < ActiveRecord::Base
  validates :email, :confirmation => true
  validates :email_confirmation, :presence => true
end
```

The default error message for this helper is "doesn't match confirmation".

3.3.4 `exclusion`

This helper validates that the attributes' values are not included in a given set. In fact, this set can be any enumerable object.

```
class Account < ActiveRecord::Base
  validates :subdomain, :exclusion => { :in => %w(www us ca jp),
    :message => "Subdomain %{value} is reserved." }
end
```

The `exclusion` helper has an option `:in` that receives the set of values that will not be accepted for the validated attributes. The `:in` option has an alias called `:within` that you can use for the same purpose, if you'd like to. This example uses the `:message` option to show how you can include the attribute's value.

The default error message is "is reserved".

3.3.5 `format`

This helper validates the attributes' values by testing whether they match a given regular expression, which is specified using the `:with` option.

```
class Product < ActiveRecord::Base
  validates :legacy_code, :format => { :with => /\A[a-zA-Z]+\z/,
    :message => "Only letters allowed" }
end
```

The default error message is "is invalid".

3.3.6 `inclusion`

This helper validates that the attributes' values are included in a given set. In fact, this set can be any enumerable object.

```
class Coffee < ActiveRecord::Base
  validates :size, :inclusion => { :in => %w(small medium large),
    :message => "%{value} is not a valid size" }
end
```

The **`inclusion`** helper has an option **`:in`** that receives the set of values that will be accepted. The **`:in`** option has an alias called **`:within`** that you can use for the same purpose, if you'd like to. The previous example uses the **`:message`** option to show how you can include the attribute's value.

The default error message for this helper is "is not included in the list".

3.3.7 `length`

This helper validates the length of the attributes' values. It provides a variety of options, so you can specify length constraints in different ways:

```
class Person < ActiveRecord::Base
  validates :name, :length => { :minimum => 2 }
  validates :bio, :length => { :maximum => 500 }
  validates :password, :length => { :in => 6..20 }
  validates :registration_number, :length => { :is => 6 }
end
```

The possible length constraint options are:

- **`:minimum`** – The attribute cannot have less than the specified length.
- **`:maximum`** – The attribute cannot have more than the specified length.

- `:in` (or `:within`) – The attribute length must be included in a given interval. The value for this option must be a range.
- `:is` – The attribute length must be equal to the given value.

The default error messages depend on the type of length validation being performed. You can personalize these messages using the `:wrong_length`, `:too_long`, and `:too_short` options and `%{count}` as a placeholder for the number corresponding to the length constraint being used. You can still use the `:message` option to specify an error message.

```
class Person < ActiveRecord::Base
  validates :bio, :length => { :maximum => 1000,
    :too_long => "%{count} characters is the maximum allowed" }
end
```

This helper counts characters by default, but you can split the value in a different way using the `:tokenizer` option:

```
class Essay < ActiveRecord::Base
  validates :content, :length => {
    :minimum   => 300,
    :maximum   => 400,
    :tokenizer => lambda { |str| str.scan(/\w+/) },
    :too_short => "must have at least %{count} words",
    :too_long  => "must have at most %{count} words"
  }
end
```

Note that the default error messages are plural (e.g., "is too short (minimum is %{count} characters)"). For this reason, when `:minimum` is 1 you should provide a personalized message or use `validates_presence_of` instead. When `:in` or `:within` have a lower limit of 1, you should either provide a personalized message or call `presence` prior to `length`.

The `size` helper is an alias for `length`.

3.3.8 `numericality`

This helper validates that your attributes have only numeric values. By default, it will match an optional sign followed by an integral or floating point number. To specify that only integral numbers are allowed set `:only_integer` to true.

If you set `:only_integer` to `true`, then it will use the

```
/\A[+-]?\d+\Z/
```

regular expression to validate the attribute's value. Otherwise, it will try to convert the value to a number using `Float`.

Note that the regular expression above allows a trailing newline character.

```
class Player < ActiveRecord::Base
  validates :points, :numericality => true
  validates :games_played, :numericality => { :only_integer => true }
end
```

Besides `:only_integer`, this helper also accepts the following options to add constraints to acceptable values:

- `:greater_than` – Specifies the value must be greater than the supplied value. The default error message for this option is "must be greater than %{count}".
- `:greater_than_or_equal_to` – Specifies the value must be greater than or equal to the supplied value. The default error message for this option is "must be greater than or equal to %{count}".
- `:equal_to` – Specifies the value must be equal to the supplied value. The default error message for this option is "must be equal to %{count}".

- **:less_than** – Specifies the value must be less than the supplied value. The default error message for this option is "must be less than %{count}".
- **:less_than_or_equal_to** – Specifies the value must be less than or equal the supplied value. The default error message for this option is "must be less than or equal to %{count}".
- **:odd** – Specifies the value must be an odd number if set to true. The default error message for this option is "must be odd".
- **:even** – Specifies the value must be an even number if set to true. The default error message for this option is "must be even".

The default error message is "is not a number".

3.3.9 presence

This helper validates that the specified attributes are not empty. It uses the `blank?` method to check if the value is either `nil` or a blank string, that is, a string that is either empty or consists of whitespace.

```
class Person < ActiveRecord::Base
  validates :name, :login, :email, :presence => true
end
```

If you want to be sure that an association is present, you'll need to test whether the foreign key used to map the association is present, and not the associated object itself.

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order_id, :presence => true
end
```

Since `false.blank?` is true, if you want to validate the presence of a boolean field you should use `validates :field_name, :inclusion => { :in => [true, false] }`.

The default error message is "can't be empty".

3.3.10 `uniqueness`

This helper validates that the attribute's value is unique right before the object gets saved. It does not create a uniqueness constraint in the database, so it may happen that two different database connections create two records with the same value for a column that you intend to be unique. To avoid that, you must create a unique index in your database.

```
class Account < ActiveRecord::Base
  validates :email, :uniqueness => true
end
```

The validation happens by performing an SQL query into the model's table, searching for an existing record with the same value in that attribute.

There is a `:scope` option that you can use to specify other attributes that are used to limit the uniqueness check:

```
class Holiday < ActiveRecord::Base
  validates :name, :uniqueness => { :scope => :year,
    :message => "should happen once per year" }
end
```

There is also a `:case_sensitive` option that you can use to define whether the uniqueness constraint will be case sensitive or not. This option defaults to true.

```
class Person < ActiveRecord::Base
  validates :name, :uniqueness => { :case_sensitive => false }
end
```

Note that some databases are configured to perform case-insensitive searches anyway.

The default error message is "has already been taken".

3.3.11 validates_with

This helper passes the record to a separate class for validation.

```
class Person < ActiveRecord::Base
  validates_with GoodnessValidator
end

class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end
```

Errors added to `record.errors[:base]` relate to the state of the record as a whole, and not to a specific attribute.

The `validates_with` helper takes a class, or a list of classes to use for validation. There is no default error message for `validates_with`. You must manually add errors to the record's errors collection in the validator class.

To implement the validate method, you must have a `record` parameter defined, which is the record to be validated.

Like all other validations, `validates_with` takes the `:if`, `:unless` and `:on` options. If you pass any other options, it will send those options to the validator class as `options`:

```
class Person < ActiveRecord::Base
  validates_with GoodnessValidator, :fields => [:first_name, :last_name]
end

class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if options[:fields].any?{|field| record.send(field) == "Evil" }
      record.errors[:base] << "This person is evil"
    end
  end
end
```

3.3.12 `validates_each`

This helper validates attributes against a block. It doesn't have a predefined validation function. You should create one using a block, and every attribute passed to `validates_each` will be tested against it. In the following example, we don't want names and surnames to begin with lower case.

```
class Person < ActiveRecord::Base
  validates_each :name, :surname do |record, attr, value|
record.errors.add(attr, 'must start with upper case') if value =~ /\A[a-z]/
  end
end
```

The block receives the record, the attribute's name and the attribute's value. You can do anything you like to check for valid data within the block. If your validation fails, you should add an error message to the model, therefore making it invalid.

3.4 Common Validation Options

These are common validation options:

3.4.1 `:allow_nil`

The `:allow_nil` option skips the validation when the value being validated is `nil`.

```
class Coffee < ActiveRecord::Base
  validates :size, :inclusion => { :in => %w(small medium large),
    :message => "%{value} is not a valid size" }, :allow_nil => true
end
```

`:allow_nil` is ignored by the presence validator.

3.4.2 `:allow_blank`

The `:allow_blank` option is similar to the `:allow_nil` option. This option will let validation pass if the attribute's value is `blank?`, like `nil` or an empty string for example.

```
class Topic < ActiveRecord::Base
  validates :title, :length => { :is => 5 }, :allow_blank => true
end

Topic.create("title" => "").valid?  # => true
Topic.create("title" => nil).valid? # => true
```

`:allow_blank` is ignored by the presence validator.

3.4.3 `:message`

As you've already seen, the `:message` option lets you specify the message that will be added to the `errors` collection when validation fails. When this option is not used, Active Record will use the respective default error message for each validation helper.

3.4.4 `:on`

The `:on` option lets you specify when the validation should happen. The default behavior for all the built-in validation helpers is to be run on save (both when you're creating a new record and when you're updating it). If you want to change it, you can use `:on => :create` to run the validation only when a new record is created or `:on => :update` to run the validation only when a record is updated.

```
class Person < ActiveRecord::Base
  # it will be possible to update email with a duplicated value
  validates :email, :uniqueness => true, :on => :create

  # it will be possible to create the record with a non-numerical age
  validates :age, :numericality => true, :on => :update

  # the default (validates on both create and update)
  validates :name, :presence => true, :on => :save
end
```

3.5 Conditional Validation

Sometimes it will make sense to validate an object just when a given predicate is satisfied. You can do that by using the `:if` and `:unless` options, which can take a symbol, a string or a `Proc`. You may use the `:if` option when you want to specify when the validation **should** happen. If you want to specify when the validation **should not** happen, then you may use the `:unless` option.

3.5.1 Using a Symbol with `:if` and `:unless`

You can associate the `:if` and `:unless` options with a symbol corresponding to the name of a method that will get called right before validation happens. This is the most commonly used option.

```
class Order < ActiveRecord::Base
  validates :card_number, :presence => true, :if => :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

3.5.2 Using a String with `:if` and `:unless`

You can also use a string that will be evaluated using `eval` and needs to contain valid Ruby code. You should use this option only when the string represents a really short condition.

```
class Person < ActiveRecord::Base
  validates :surname, :presence => true, :if => "name.nil?"
end
```

3.5.3 Using a Proc with `:if` and `:unless`

Finally, it's possible to associate `:if` and `:unless` with a `Proc` object which will be called. Using a `Proc` object gives you the ability to write an inline condition instead of a separate method. This option is best suited for one-liners.

```
class Account < ActiveRecord::Base
  validates :password, :confirmation => true,
    :unless => Proc.new { |a| a.password.blank? }
end
```

3.5.4 Grouping conditional validations

Sometimes it is useful to have multiple validations use one condition, it can be easily achieved using `with_options`.

```
class User < ActiveRecord::Base
  with_options :if => :is_admin? do |admin|
    admin.validates :password, :length => { :minimum => 10 }
    admin.validates :email, :presence => true
  end
end
```

All validations inside of `with_options` block will have automatically passed the condition `:if => :is_admin?`

3.6 Performing Custom Validations

When the built-in validation helpers are not enough for your needs, you can write your own validators or validation methods as you prefer.

3.6.1 Custom Validators

Custom validators are classes that extend `ActiveModel::Validator`. These classes must implement a `validate` method which takes a record as an argument and performs the validation on it. The custom validator is called using the `validates_with` method.

```
class MyValidator < ActiveModel::Validator
  def validate(record)
    unless record.name.starts_with? 'X'
      record.errors[:name] << 'Need a name starting with X please!'
    end
  end
end

class Person
  include ActiveModel::Validations
  validates_with MyValidator
end
```

The easiest way to add custom validators for validating individual attributes is with the convenient `ActiveModel::EachValidator`. In this case, the custom validator class must implement a `validate_each` method which takes three arguments: record, attribute and value which correspond to the instance, the attribute to be validated and the value of the attribute in the passed instance.

```
class EmailValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
      record.errors[attribute] << (options[:message] || "is not an email")
    end
  end
end

class Person < ActiveRecord::Base
  validates :email, :presence => true, :email => true
end
```

As shown in the example, you can also combine standard validations with your own custom validators.

3.6.2 Custom Methods

You can also create methods that verify the state of your models and add messages to the `errors` collection when they are invalid. You must then register these methods by using the `validate` class method, passing in the symbols for the validation methods' names.

You can pass more than one symbol for each class method and the respective validations will be run in the same order as they were registered.

```
class Invoice < ActiveRecord::Base
  validate :expiration_date_cannot_be_in_the_past,
    :discount_cannot_be_greater_than_total_value

  def expiration_date_cannot_be_in_the_past
    if !expiration_date.blank? and expiration_date < Date.today
      errors.add(:expiration_date, "can't be in the past")
    end
  end

  def discount_cannot_be_greater_than_total_value
    if discount > total_value
      errors.add(:discount, "can't be greater than total value")
    end
  end
end
```

By default such validations will run every time you call `valid?`. It is also possible to control when to run these custom validations by giving an `:on` option to the `validate` method, with either: `:create` or `:update`.

```
class Invoice < ActiveRecord::Base
  validate :active_customer, :on => :create

  def active_customer
    errors.add(:customer_id, "is not active") unless customer.active?
  end
end
```

You can even create your own validation helpers and reuse them in several different models. For example, an application that manages surveys may find it useful to express that a certain field corresponds to a set of choices:

```
ActiveRecord::Base.class_eval do
  def self.validates_as_choice(attr_name, n, options={})
    validates attr_name, :inclusion => { { :in => 1..n }.merge!(options) }
  end
end
```

Simply reopen `ActiveRecord::Base` and define a class method like that. You'd typically put this code somewhere in `config/initializers`. You can use this helper like this:

```
class Movie < ActiveRecord::Base
  validates_as_choice :rating, 5
end
```

3.7 Working with Validation Errors

In addition to the `valid?` and `invalid?` methods covered earlier, Rails provides a number of methods for working with the `errors` collection and inquiring about the validity of objects.

The following is a list of the most commonly used methods. Please refer to the `ActiveModel::Errors` documentation for a list of all the available methods.

3.7.1 errors

Returns an instance of the class `ActiveModel::Errors` (which behaves like an ordered hash) containing all errors. Each key is the attribute name and the value is an array of strings with all errors.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new
person.valid? # => false
person.errors
 # => {:name => ["can't be blank", "is too short (minimum is 3 characters)"

person = Person.new(:name => "John Doe")
person.valid? # => true
person.errors # => []
```

3.7.2 errors[]

`errors[]` is used when you want to check the error messages for a specific attribute. It returns an array of strings with all error messages for the given attribute, each string with one error message. If there are no errors related to the attribute, it returns an empty array.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new(:name => "John Doe")
person.valid? # => true
person.errors[:name] # => []

person = Person.new(:name => "JD")
person.valid? # => false
person.errors[:name] # => ["is too short (minimum is 3 characters)"]

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 characters)"]
```

3.7.3 `errors.add`

The `add` method lets you manually add messages that are related to particular attributes. You can use the `errors.full_messages` or `errors.to_a` methods to view the messages in the form they might be displayed to a user. Those particular messages get the attribute name prepended (and capitalized). `add` receives the name of the attribute you want to add the message to, and the message itself.

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors.add(:name, "cannot contain the characters !@#%*()_-+=")
  end
end

person = Person.create(:name => "!@#")

person.errors[:name]
 # => ["cannot contain the characters !@#%*()_-+="]

person.errors.full_messages
 # => ["Name cannot contain the characters !@#%*()_-+="]
```

Another way to do this is using `[]=` setter

```
class Person < ActiveRecord::Base
    def a_method_used_for_validation_purposes
      errors[:name] = "cannot contain the characters !@#%*()_-+="
    end
  end

  person = Person.create(:name => "!@#")

  person.errors[:name]
   # => ["cannot contain the characters !@#%*()_-+="]

  person.errors.to_a
   # => ["Name cannot contain the characters !@#%*()_-+="]
```

3.7.4 `errors[:base]`

You can add error messages that are related to the object's state as a whole, instead of being related to a specific attribute. You can use this method when you want to say that the object is invalid, no matter the values of its attributes. Since **`errors[:base]`** is an array, you can simply add a string to it and it will be used as an error message.

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors[:base] << "This person is invalid because ..."
  end
end
```

3.7.5 `errors.clear`

The `clear` method is used when you intentionally want to clear all the messages in the `errors` collection. Of course, calling `errors.clear` upon an invalid object won't actually make it valid: the `errors` collection will now be empty, but the next time you call `valid?` or any method that tries to save this object to the database, the validations will run again. If any of the validations fail, the `errors` collection will be filled again.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 characters)"]

person.errors.clear
person.errors.empty? # => true

p.save # => false

p.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 characters)"]
```

3.7.6 `errors.size`

The `size` method returns the total number of error messages for the object.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new
person.valid? # => false
person.errors.size # => 2

person = Person.new(:name => "Andrea", :email => "andrea@example.com")
person.valid? # => true
person.errors.size # => 0
```

3.8 Displaying Validation Errors in the View

DynamicForm provides helpers to display the error messages of your models in your view templates.

You can install it as a gem by adding this line to your Gemfile:

```
gem "dynamic_form"
```

Now you will have access to the two helper methods `error_messages` and `error_messages_for` in your view templates.

3.8.1 `error_messages` and `error_messages_for`

When creating a form with the `form_for` helper, you can use the `error_messages` method on the form builder to render all failed validation messages for the current model instance.

```
class Product < ActiveRecord::Base
  validates :description, :value, :presence => true
  validates :value, :numericality => true, :allow_nil => true
end
```

```
<%= form_for(@product) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :description %><br />
    <%= f.text_field :description %>
  </p>
  <p>
    <%= f.label :value %><br />
    <%= f.text_field :value %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
```

If you submit the form with empty fields, the result will be similar to the one shown below:

`../error_messages.png`

The appearance of the generated HTML will be different from the one shown, unless you have used scaffolding. See Customizing the Error Messages CSS.

You can also use the `error_messages_for` helper to display the error messages of a model assigned to a view template. It is very similar to the previous example and will achieve exactly the same result.

```
<%= error_messages_for :product %>
```

The displayed text for each error message will always be formed by the capitalized name of the attribute that holds the error, followed by the error message itself.

Both the `form.error_messages` and the `error_messages_for` helpers accept options that let you customize the `div` element that holds the messages, change the header text, change the message below the header, and specify the tag used for the header element. For example,

```
<%= f.error_messages :header_message => "Invalid product!",
  :message => "You'll need to fix the following fields:",
  :header_tag => :h3 %>
```

results in:

`../customized_error_messages.png`

If you pass `nil` in any of these options, the corresponding section of the `div` will be discarded.

3.8.2 Customizing the Error Messages CSS

The selectors used to customize the style of error messages are:

- `.field_with_errors` – Style for the form fields and labels with errors.

- `#error_explanation` – Style for the `div` element with the error messages.
- `#error_explanation h2` – Style for the header of the `div` element.
- `#error_explanation p` – Style for the paragraph holding the message that appears right below the header of the `div` element.
- `#error_explanation ul li` – Style for the list items with individual error messages.

If scaffolding was used, file `app/assets/stylesheets/scaffolds.css.sc` will have been generated automatically. This file defines the red-based styles you saw in the examples above.

The name of the class and the id can be changed with the `:class` and `:id` options, accepted by both helpers.

3.8.3 Customizing the Error Messages HTML

By default, form fields with errors are displayed enclosed by a `div` element with the `field_with_errors`CSS class. However, it's possible to override that.

The way form fields with errors are treated is defined by `ActionView::Ba` This is a `Proc` that receives two parameters:

- A string with the HTML tag
- An instance of `ActionView::Helpers::InstanceTag`.

Below is a simple example where we change the Rails behavior to always display the error messages in front of each of the form fields in error. The error messages will be enclosed by a `span` element with a `validation-error`CSS class. There will be no `div` element enclosing the `input` element, so we get rid of that red border around the text field. You can use the `validation-error`CSS class to style it anyway you want.

```
ActionView::Base.field_error_proc = Proc.new do |html_tag, instance|
errors = Array(instance.error_message).join(',')
%(#{html_tag}<span class="validation-error">&nbsp;#{errors}</span>).html_sa
end
```

The result looks like the following:

../validation_error_messages.png

3.9 Callbacks Overview

Callbacks are methods that get called at certain moments of an object's life cycle. With callbacks it is possible to write code that will run whenever an Active Record object is created, saved, updated, deleted, validated, or loaded from the database.

3.9.1 Callback Registration

In order to use the available callbacks, you need to register them. You can implement the callbacks as ordinary methods and use a macro-style class method to register them as callbacks:

```
class User < ActiveRecord::Base
  validates :login, :email, :presence => true

  before_validation :ensure_login_has_a_value

  protected
  def ensure_login_has_a_value
    if login.nil?
      self.login = email unless email.blank?
    end
  end
end
```

The macro-style class methods can also receive a block. Consider using this style if the code inside your block is so short that it fits in a single line:

```
class User < ActiveRecord::Base
  validates :login, :email, :presence => true

  before_create do |user|
    user.name = user.login.capitalize if user.name.blank?
  end
end
```

It is considered good practice to declare callback methods as protected or private. If left public, they can be called from outside of the model and violate the principle of object encapsulation.

3.10 Available Callbacks

Here is a list with all the available Active Record callbacks, listed in the same order in which they will get called during the respective operations:

3.10.1 Creating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_create`
- `around_create`
- `after_create`
- `after_save`

3.10.2 Updating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`

- `before_update`
- `around_update`
- `after_update`
- `after_save`

3.10.3 Destroying an Object

- `before_destroy`
- `around_destroy`
- `after_destroy`

`after_save` runs both on create and update, but always after the more specific callbacks `after_create` and `after_update`, no matter the order in which the macro calls were executed.

3.10.4 `after_initialize` and `after_find`

The `after_initialize` callback will be called whenever an Active Record object is instantiated, either by directly using `new` or when a record is loaded from the database. It can be useful to avoid the need to directly override your Active Record `initialize` method.

The `after_find` callback will be called whenever Active Record loads a record from the database. `after_find` is called before `after_initiali` if both are defined.

The `after_initialize` and `after_find` callbacks have no `before_*` counterparts, but they can be registered just like the other Active Record callbacks.

```
class User < ActiveRecord::Base
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
  end
end

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

3.11 Running Callbacks

The following methods trigger callbacks:

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(:validate => false)`
- `toggle!`

- `update`
- `update_attribute`
- `update_attributes`
- `update_attributes!`
- `valid?`

Additionally, the `after_find` callback is triggered by the following finder methods:

- `all`
- `first`
- `find`
- `find_all_by_attribute`
- `find_by_attribute`
- `find_by_attribute!`
- `last`

The `after_initialize` callback is triggered every time a new object of the class is initialized.

3.12 Skipping Callbacks

Just as with validations, it is also possible to skip callbacks. These methods should be used with caution, however, because important business rules and application logic may be kept in callbacks. Bypassing them without understanding the potential implications may lead to invalid data.

- `decrement`
- `decrement_counter`
- `delete`

- `delete_all`
- `find_by_sql`
- `increment`
- `increment_counter`
- `toggle`
- `touch`
- `update_column`
- `update_all`
- `update_counters`

3.13 Halting Execution

As you start registering new callbacks for your models, they will be queued for execution. This queue will include all your model's validations, the registered callbacks, and the database operation to be executed.

The whole callback chain is wrapped in a transaction. If any before callback method returns exactly `false` or raises an exception, the execution chain gets halted and a ROLLBACK is issued; after callbacks can only accomplish that by raising an exception.

Raising an arbitrary exception may break code that expects `save` and its friends not to fail like that. The `ActiveRecord::Rollback` exception is thought precisely to tell Active Record a rollback is going on. That one is internally captured but not reraised.

3.14 Relational Callbacks

Callbacks work through model relationships, and can even be defined by them. Suppose an example where a user has many posts.

A user's posts should be destroyed if the user is destroyed. Let's add an `after_destroy` callback to the `User` model by way of its relationship to the `Post` model:

```
class User < ActiveRecord::Base
  has_many :posts, :dependent => :destroy
end

class Post < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Post destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.posts.create!
=> #<Post id: 1, user_id: 1>
>> user.destroy
Post destroyed
=> #<User id: 1>
```

3.15 Conditional Callbacks

As with validations, we can also make the calling of a callback method conditional on the satisfaction of a given predicate. We can do this using the `:if` and `:unless` options, which can take a symbol, a string or a `Proc`. You may use the `:if` option when you want to specify under which conditions the callback **should** be called. If you want to specify the conditions under which the callback **should not** be called, then you may use the `:unless` option.

3.15.1 Using `:if` and `:unless` with a `Symbol`

You can associate the `:if` and `:unless` options with a symbol corresponding to the name of a predicate method that will get called right before the callback. When using the `:if` option, the callback won't be executed if the predicate method returns false; when using the `:unless` option, the callback won't be executed if the predicate method returns true. This is the most common option. Using this form of registration it is also possible to register several different predicates that should be called to check if the callback should be executed.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, :if => :paid_with_card?
end
```

3.15.2 Using `:if` and `:unless` with a String

You can also use a string that will be evaluated using `eval` and hence needs to contain valid Ruby code. You should use this option only when the string represents a really short condition:

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, :if => "paid_with_card?"
end
```

3.15.3 Using `:if` and `:unless` with a `Proc`

Finally, it is possible to associate `:if` and `:unless` with a `Proc` object. This option is best suited when writing short validation methods, usually one-liners:

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number,
    :if => Proc.new { |order| order.paid_with_card? }
end
```

3.15.4 Multiple Conditions for Callbacks

When writing conditional callbacks, it is possible to mix both **:if** and **:unless** in the same callback declaration:

```
class Comment < ActiveRecord::Base
  after_create :send_email_to_author, :if => :author_wants_emails?,
    :unless => Proc.new { |comment| comment.post.ignore_comments? }
end
```

3.16 Callback Classes

Sometimes the callback methods that you'll write will be useful enough to be reused by other models. Active Record makes it possible to create classes that encapsulate the callback methods, so it becomes very easy to reuse them.

Here's an example where we create a class with an **after_destroy** callback for a **PictureFile** model:

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exists?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

When declared inside a class, as above, the callback methods will receive the model object as a parameter. We can now use the callback class in the model:

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks.new
end
```

Note that we needed to instantiate a new `PictureFileCallbacks` object, since we declared our callback as an instance method. This is particularly useful if the callbacks make use of the state of the instantiated object. Often, however, it will make more sense to declare the callbacks as class methods:

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exists?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

If the callback method is declared this way, it won't be necessary to instantiate a `PictureFileCallbacks` object.

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks
end
```

You can declare as many callbacks as you want inside your callback classes.

3.17 Observers

Observers are similar to callbacks, but with important differences. Whereas callbacks can pollute a model with code that isn't directly related to its purpose, observers allow you to add the same functionality without changing the code of the model. For example, it could be argued that a `User` model should not include code to send registration confirmation emails. Whenever you use callbacks with code that isn't directly related to your model, you may want to consider creating an observer instead.

3.17.1 Creating Observers

For example, imagine a `User` model where we want to send an email every time a new user is created. Because sending emails is not directly related to our model's purpose, we should create an observer to contain the code implementing this functionality.

```
$ rails generate observer User
```

generates `app/models/user_observer.rb` containing the observer class `UserObserver`:

```
class UserObserver < ActiveRecord::Observer
end
```

You may now add methods to be called at the desired occasions:

```
class UserObserver < ActiveRecord::Observer
  def after_create(model)
    # code to send confirmation email...
  end
end
```

As with callback classes, the observer's methods receive the observed model as a parameter.

3.17.2 Registering Observers

Observers are conventionally placed inside of your app/models directory and registered in your application's config/application.rb file. For example, the UserObserver above would be saved as app/models/use
and registered in config/application.rb this way:

```
# Activate observers that should always be running.
config.active_record.observers = :user_observer
```

As usual, settings in config/environments take precedence over those in config/application.rb. So, if you prefer that an observer doesn't run in all environments, you can simply register it in a specific environment instead.

3.17.3 Sharing Observers

By default, Rails will simply strip "Observer" from an observer's name to find the model it should observe. However, observers can also be used to add behavior to more than one model, and thus it is possible to explicitly specify the models that our observer should observe:

```
class MailerObserver < ActiveRecord::Observer
  observe :registration, :user

  def after_create(model)
    # code to send confirmation email...
  end
end
```

In this example, the after_create method will be called whenever a Registration or User is created. Note that this new MailerObserver would also need to be registered in config/application.rb in order to take effect:

```
# Activate observers that should always be running.
config.active_record.observers = :mailer_observer
```

3.18 Transaction Callbacks

There are two additional callbacks that are triggered by the completion of a database transaction: `after_commit` and `after_rollback`. These callbacks are very similar to the `after_save` callback except that they don't execute until after database changes have either been committed or rolled back. They are most useful when your active record models need to interact with external systems which are not part of the database transaction.

Consider, for example, the previous example where the `PictureFile` model needs to delete a file after the corresponding record is destroyed. If anything raises an exception after the `after_destroy` callback is called and the transaction rolls back, the file will have been deleted and the model will be left in an inconsistent state. For example, suppose that `picture_file_2` in the code below is not valid and the `save!` method raises an error.

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

By using the `after_commit` callback we can account for this case.

```
class PictureFile < ActiveRecord::Base
  attr_accessor :delete_file

  after_destroy do |picture_file|
    picture_file.delete_file = picture_file.filepath
  end

  after_commit do |picture_file|
    if picture_file.delete_file && File.exist?(picture_file.delete_file)
      File.delete(picture_file.delete_file)
      picture_file.delete_file = nil
    end
  end
end
```

The **`after_commit`** and **`after_rollback`** callbacks are guaranteed to be called for all models created, updated, or destroyed within a transaction block. If any exceptions are raised within one of these callbacks, they will be ignored so that they don't interfere with the other callbacks. As such, if your callback code could raise an exception, you'll need to rescue it and handle it appropriately within the callback.

Chapter 4

A Guide to Active Record Associations

This guide covers the association features of Active Record. By referring to this guide, you will be able to:

- Declare associations between Active Record models
- Understand the various types of Active Record associations
- Use the methods added to your models by creating associations

4.1 Why Associations?

Why do we need associations between models? Because they make common operations simpler and easier in your code. For example, consider a simple Rails application that includes a model for customers and a model for orders. Each customer can have many orders. Without associations, the model declarations would look like this:

```
class Customer < ActiveRecord::Base
end

class Order < ActiveRecord::Base
end
```

Now, suppose we wanted to add a new order for an existing customer. We'd need to do something like this:

```
@order = Order.create(:order_date => Time.now,
  :customer_id => @customer.id)
```

Or consider deleting a customer, and ensuring that all of its orders get deleted as well:

```
@orders = Order.where(:customer_id => @customer.id)
@orders.each do |order|
  order.destroy
end
@customer.destroy
```

With Active Record associations, we can streamline these — and other — operations by declaratively telling Rails that there is a connection between the two models. Here's the revised code for setting up customers and orders:

```
class Customer < ActiveRecord::Base
  has_many :orders, :dependent => :destroy
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

With this change, creating a new order for a particular customer is easier:

```
@order = @customer.orders.create(:order_date => Time.now)
```

Deleting a customer and all of its orders is <u>much</u> easier:

```
@customer.destroy
```

To learn more about the different types of associations, read the next section of this guide. That's followed by some tips and tricks for working with associations, and then by a complete reference to the methods and options for associations in Rails.

4.2 The Types of Associations

In Rails, an <u>association</u> is a connection between two Active Record models. Associations are implemented using macro-style calls, so that you can declaratively add features to your models. For example, by declaring that one model `belongs_to` another, you instruct Rails to maintain Primary Key–Foreign Key information between instances of the two models, and you also get a number of utility methods added to your model. Rails supports six types of associations:

- `belongs_to`
- `has_one`
- `has_many`
- `has_many :through`
- `has_one :through`
- `has_and_belongs_to_many`

In the remainder of this guide, you'll learn how to declare and use the various forms of associations. But first, a quick introduction to the situations where each association type is appropriate.

4.2.1 The belongs_to Association

A **belongs_to** association sets up a one-to-one connection with another model, such that each instance of the declaring model "belongs to" one instance of the other model. For example, if your application includes customers and orders, and each order can be assigned to exactly one customer, you'd declare the order model this way:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

../belongs_to.png

4.2.2 The has_one Association

A **has_one** association also sets up a one-to-one connection with another model, but with somewhat different semantics (and consequences). This association indicates that each instance of a model contains or possesses one instance of another model. For example, if each supplier in your application has only one account, you'd declare the supplier model like this:

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

../has_one.png

4.2.3 The has_many Association

A **has_many** association indicates a one-to-many connection with another model. You'll often find this association on the "other side" of a **belongs_to** association. This association indicates that each instance of the model has zero or more instances of another model. For example, in an application containing customers and orders, the customer model could be declared like this:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

The name of the other model is pluralized when declaring a **has_many** association.

../has_many.png

4.2.4 The has_many :through Association

A **has_many :through** association is often used to set up a many-to-many connection with another model. This association indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model. For example, consider a medical practice where patients make appointments to see physicians. The relevant association declarations could look like this:

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

`../has_many_through.png`

The collection of join models can be managed via the API. For example, if you assign

```
physician.patients = patients
```

new join models are created for newly associated objects, and if some are gone their rows are deleted.

Automatic deletion of join models is direct, no destroy callbacks are triggered.

The **has_many :through** association is also useful for setting up "shortcuts" through nested **has_many** associations. For example, if a document has many sections, and a section has many paragraphs, you may sometimes want to get a simple collection of all paragraphs in the document. You could set that up this way:

```
class Document < ActiveRecord::Base
  has_many :sections
  has_many :paragraphs, :through => :sections
end

class Section < ActiveRecord::Base
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ActiveRecord::Base
  belongs_to :section
end
```

With **:through => :sections** specified, Rails will now understand:

```
@document.paragraphs
```

4.2.5 The has_one :through Association

A **has_one :through** association sets up a one-to-one connection with another model. This association indicates that the declaring model can be matched with one instance of another model by proceeding through a third model. For example, if each supplier has one account, and each account is associated with one account history, then the customer model could look like this:

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

../has_one_through.png

4.2.6 The `has_and_belongs_to_many` Association

A `has_and_belongs_to_many` association creates a direct many-to-many connection with another model, with no intervening model. For example, if your application includes assemblies and parts, with each assembly having many parts and each part appearing in many assemblies, you could declare the models this way:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

../habtm.png

4.2.7 Choosing Between `belongs_to` and `has_one`

If you want to set up a one-to-one relationship between two models, you'll need to add `belongs_to` to one, and `has_one` to the other. How do you know which is which?

The distinction is in where you place the foreign key (it goes on the table for the class declaring the **`belongs_to`** association), but you should give some thought to the actual meaning of the data as well. The **`has_one`** relationship says that one of something is yours – that is, that something points back to you. For example, it makes more sense to say that a supplier owns an account than that an account owns a supplier. This suggests that the correct relationships are like this:

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
end
```

The corresponding migration might look like this:

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string  :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps
    end
  end
end
```

Using **`t.integer :supplier_id`** makes the foreign key naming obvious and explicit. In current versions of Rails, you can abstract away this implementation detail by using **`t.references :supplier`** instead.

4.2.8 Choosing Between has_many :through and has_and_belongs_to_many

Rails offers two different ways to declare a many-to-many relationship between models. The simpler way is to use `has_and_belongs_to_many`, which allows you to make the association directly:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

The second way to declare a many-to-many relationship is to use `has_many :through`. This makes the association indirectly, through a join model:

```
class Assembly < ActiveRecord::Base
  has_many :manifests
  has_many :parts, :through => :manifests
end

class Manifest < ActiveRecord::Base
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord::Base
  has_many :manifests
  has_many :assemblies, :through => :manifests
end
```

The simplest rule of thumb is that you should set up a `has_many :through` relationship if you need to work with the relationship model as an independent entity. If you don't need to do anything with the relationship model, it may be simpler to set up a `has_and_belongs_to_many` relationship (though you'll need to remember to create the joining table in the database).

You should use `has_many :through` if you need validations, callbacks, or extra attributes on the join model.

4.2.9 Polymorphic Associations

A slightly more advanced twist on associations is the polymorphic association. With polymorphic associations, a model can belong to more than one other model, on a single association. For example, you might have a picture model that belongs to either an employee model or a product model. Here's how this could be declared:

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

You can think of a polymorphic `belongs_to` declaration as setting up an interface that any other model can use. From an instance of the `Employee` model, you can retrieve a collection of pictures: `@employee.pictures`.

Similarly, you can retrieve `@product.pictures`.

If you have an instance of the `Picture` model, you can get to its parent via `@picture.imageable`. To make this work, you need to declare both a foreign key column and a type column in the model that declares the polymorphic interface:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end
  end
end
```

This migration can be simplified by using the `t.references` form:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, :polymorphic => true
      t.timestamps
    end
  end
end
```

../polymorphic.png

4.2.10 Self Joins

In designing a data model, you will sometimes find a model that should have a relation to itself. For example, you may want to store all employees in a single database model, but be able to trace relationships such as between manager and subordinates. This situation can be modeled with self-joining associations:

```
class Employee < ActiveRecord::Base
  has_many :subordinates, :class_name => "Employee",
    :foreign_key => "manager_id"
  belongs_to :manager, :class_name => "Employee"
end
```

With this setup, you can retrieve `@employee.subordinates` and `@employee.manager`.

4.3 Tips, Tricks, and Warnings

Here are a few things you should know to make efficient use of Active Record associations in your Rails applications:

- Controlling caching
- Avoiding name collisions
- Updating the schema
- Controlling association scope
- Bi-directional associations

4.3.1 Controlling Caching

All of the association methods are built around caching, which keeps the result of the most recent query available for further operations. The cache is even shared across methods. For example:

```
customer.orders        # retrieves orders from the database
customer.orders.size   # uses the cached copy of orders
customer.orders.empty? # uses the cached copy of orders
```

But what if you want to reload the cache, because data might have been changed by some other part of the application? Just pass

true to the association call:

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
customer.orders(true).empty?# discards the cached copy of orders
                                # and goes back to the database
```

4.3.2 Avoiding Name Collisions

You are not free to use just any name for your associations. Because creating an association adds a method with that name to the model, it is a bad idea to give an association a name that is already used for an instance method of `ActiveRecord::Base`. The association method would override the base method and break things. For instance, **attributes** or **connection** are bad names for associations.

4.3.3 Updating the Schema

Associations are extremely useful, but they are not magic. You are responsible for maintaining your database schema to match your associations. In practice, this means two things, depending on what sort of associations you are creating. For `belongs_to` associations you need to create foreign keys, and for `has_and_belongs_to_many` associations you need to create the appropriate join table.

Creating Foreign Keys for belongs_to Associations

When you declare a `belongs_to` association, you need to create foreign keys as appropriate. For example, consider this model:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

This declaration needs to be backed up by the proper foreign key declaration on the orders table:

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :orders do |t|
      t.datetime :order_date
      t.string   :order_number
      t.integer  :customer_id
    end
  end
end
```

If you create an association some time after you build the underlying model, you need to remember to create an `add_column` migration to provide the necessary foreign key.

Creating Join Tables for `has_and_belongs_to_many` Associations

If you create a `has_and_belongs_to_many` association, you need to explicitly create the joining table. Unless the name of the join table is explicitly specified by using the `:join_table` option, Active Record creates the name by using the lexical order of the class names. So a join between customer and order models will give the default join table name of "customers_orders" because "c" outranks "o" in lexical ordering.

The precedence between model names is calculated using the < operator for `String`. This means that if the strings are of different lengths, and the strings are equal when compared up to the shortest length, then the longer string is considered of higher lexical precedence than the shorter one. For example, one would expect the tables "paper_boxes" and "papers" to generate a join table name of "papers_paper_boxes" because of the length of the name "paper_boxes", but it in fact generates a join table name of "paper_boxes_papers" (because the underscore '_' is lexicographically less than 's' in common encodings).

Whatever the name, you must manually generate the join table with an appropriate migration. For example, consider these associations:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

These need to be backed up by a migration to create the `assemblies_part` table. This table should be created without a primary key:

```
class CreateAssemblyPartJoinTable < ActiveRecord::Migration
  def change
    create_table :assemblies_parts, :id => false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end
  end
end
```

We pass `:id => false` to `create_table` because that table does not represent a model. That's required for the association to work properly. If you observe any strange behavior in a `has_and_belongs_to_many` association like mangled models IDs, or exceptions about conflicting IDs chances are you forgot that bit.

4.3.4 Controlling Association Scope

By default, associations look for objects only within the current module's scope. This can be important when you declare Active

Record models within a module. For example:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account
    end

    class Account < ActiveRecord::Base
       belongs_to :supplier
    end
  end
end
```

This will work fine, because both the `Supplier` and the `Account` class are defined within the same scope. But the following will not work, because `Supplier` and `Account` are defined in different scopes:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account
    end
  end

  module Billing
    class Account < ActiveRecord::Base
       belongs_to :supplier
    end
  end
end
```

To associate a model with a model in a different namespace, you must specify the complete class name in your association declaration:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account,
        :class_name => "MyApplication::Billing::Account"
    end
  end

  module Billing
    class Account < ActiveRecord::Base
       belongs_to :supplier,
        :class_name => "MyApplication::Business::Supplier"
    end
  end
end
```

4.3.5 Bi-directional Associations

It's normal for associations to work in two directions, requiring declaration on two different models:

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

By default, Active Record doesn't know about the connection between these associations. This can lead to two copies of an object getting out of sync:

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => false
```

This happens because c and o.customer are two different in-memory representations of the same data, and neither one is automatically refreshed from changes to the other. Active Record provides the `:inverse_of` option so that you can inform it of these relations:

```
class Customer < ActiveRecord::Base
  has_many :orders, :inverse_of => :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, :inverse_of => :orders
end
```

With these changes, Active Record will only load one copy of the customer object, preventing inconsistencies and making your application more efficient:

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => true
```

There are a few limitations to `inverse_of` support:

- They do not work with `:through` associations.
- They do not work with `:polymorphic` associations.
- They do not work with `:as` associations.
- For `belongs_to` associations, `has_many` inverse associations are ignored.

4.4 Detailed Association Reference

The following sections give the details of each type of association, including the methods that they add and the options that you can use when declaring an association.

4.4.1 `belongs_to` Association Reference

The **`belongs_to`** association creates a one-to-one match with another model. In database terms, this association says that this class contains the foreign key. If the other class contains the foreign key, then you should use **`has_one`** instead.

Methods Added by `belongs_to`

When you declare a **`belongs_to`** association, the declaring class automatically gains four methods related to the association:

- `association(force_reload = false)`
- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`

In all of these methods, `association` is replaced with the symbol passed as the first argument to **`belongs_to`**. For example, given the declaration:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Each instance of the order model will have these methods:

```
customer
customer=
build_customer
create_customer
```

When initializing a new `has_one` or `belongs_to` association you must use the `build_` prefix to build the association, rather than the `association.build` method that would be used for `has_many` or `has_and_belongs_to_many` associations. To create one, use the `create_` prefix.

4.1.1.1 `association(force_reload = false)`

The `association` method returns the associated object, if any. If no associated object is found, it returns `nil`.

```
@customer = @order.customer
```

If the associated object has already been retrieved from the database for this object, the cached version will be returned. To override this behavior (and force a database read), pass `true` as the `force_reload` argument.

4.1.1.2 `association=(associate)`

The `association=` method assigns an associated object to this object. Behind the scenes, this means extracting the primary key from the associate object and setting this object's foreign key to the same value.

```
@order.customer = @customer
```

4.1.1.3 `build_association(attributes = {})`

The **`build_association`** method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through this object's foreign key will be set, but the associated object will not yet be saved.

```
@customer = @order.build_customer(:customer_number => 123,
  :customer_name => "John Doe")
```

4.1.1.4 `create_association(attributes = {})`

The **`create_association`** method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through this object's foreign key will be set, and, once it passes all of the validations specified on the associated model, the associated object will be saved.

```
@customer = @order.create_customer(:customer_number => 123,
  :customer_name => "John Doe")
```

Options for `belongs_to`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the **`belongs_to`** association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this assocation uses two such options:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :counter_cache => true,
    :conditions => "active = 1"
end
```

The `belongs_to` association supports these options:

- `:autosave`
- `:class_name`
- `:conditions`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:include`
- `:inverse_of`
- `:polymorphic`
- `:readonly`
- `:select`
- `:touch`
- `:validate`

4.1.2.1 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.1.2.2 `:class_name`

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if an order belongs to a customer, but the actual name of the model containing customers is `Patron`, you'd set things up this way:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :class_name => "Patron"
end
```

4.1.2.3 :conditions

The `:conditions` option lets you specify the conditions that the associated object must meet (in the syntax used by an SQL`WHERE` clause).

```
class Order < ActiveRecord::Base
  belongs_to :customer, :conditions => "active = 1"
end
```

4.1.2.4 :counter_cache

The `:counter_cache` option can be used to make finding the number of belonging objects more efficient. Consider these models:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

With these declarations, asking for the value of `@customer.orders.size` requires making a call to the database to perform a `COUNT(*)` query. To avoid this call, you can add a counter cache to the belonging model:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :counter_cache => true
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

With this declaration, Rails will keep the cache value up to date, and then return that value in response to the `size` method.

Although the `:counter_cache` option is specified on the model that includes the **belongs_to** declaration, the actual column must be added to the associated model. In the case above, you would need to add a column named **orders_count** to the **Customer** model. You can override the default column name if you need to:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :counter_cache => :count_of_orders
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Counter cache columns are added to the containing model's list of read-only attributes through **attr_readonly**.

4.1.2.5 :dependent

If you set the **:dependent** option to **:destroy**, then deleting this object will call the **destroy** method on the associated object to delete that object. If you set the **:dependent** option to **:delete**, then deleting this object will delete the associated object without calling its **destroy** method.

You should not specify this option on a **belongs_to** association that is connected with a **has_many** association on the other class. Doing so can lead to orphaned records in your database.

4.1.2.6 :foreign_key

By convention, Rails assumes that the column used to hold the foreign key on this model is the name of the association with the suffix **_id** added. The **:foreign_key** option lets you set the name of the foreign key directly:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :class_name => "Patron",
    :foreign_key => "patron_id"
end
```

In any case, Rails will not create foreign key columns for you. You need to explicitly define them as part of your migrations.

4.1.2.7 `:include`

You can use the `:include` option to specify second-order associations that should be eager-loaded when this association is used. For example, consider these models:

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

If you frequently retrieve customers directly from line items (`@line_item.order.customer`), then you can make your code somewhat more efficient by including customers in the association from line items to orders:

```
class LineItem < ActiveRecord::Base
  belongs_to :order, :include => :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

There's no need to use **`:include`** for immediate associations – that is, if you have **`Order belongs_to :customer`**, then the customer is eager-loaded automatically when it's needed.

4.1.2.8 `:inverse_of`

The **`:inverse_of`** option specifies the name of the **`has_many`** or **`has_one`** association that is the inverse of this association. Does not work in combination with the **`:polymorphic`** options.

```
class Customer < ActiveRecord::Base
  has_many :orders, :inverse_of => :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, :inverse_of => :orders
end
```

4.1.2.9 `:polymorphic`

Passing **`true`** to the **`:polymorphic`** option indicates that this is a polymorphic association. Polymorphic associations were discussed in detail earlier in this guide.

4.1.2.10 `:readonly`

If you set the `:readonly` option to `true`, then the associated object will be read-only when retrieved via the association.

4.1.2.11 `:select`

The `:select` option lets you override the SQL`SELECT` clause that is used to retrieve data about the associated object. By default, Rails retrieves all columns.

If you set the `:select` option on a `belongs_to` association, you should also set the `foreign_key` option to guarantee the correct results.

4.1.2.12 `:touch`

If you set the `:touch` option to `:true`, then the `updated_at` or `updated_on` timestamp on the associated object will be set to the current time whenever this object is saved or destroyed:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :touch => true
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

In this case, saving or destroying an order will update the timestamp on the associated customer. You can also specify a particular timestamp attribute to update:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :touch => :orders_updated_at
end
```

4.1.2.13 `:validate`

If you set the **`:validate`** option to **`true`**, then associated objects will be validated whenever you save this object. By default, this is **`false`**: associated objects will not be validated when this object is saved.

Do Any Associated Objects Exist?

You can see if any associated objects exist by using the association.nil? method:

```
if @order.customer.nil?
  @msg = "No customer found for this order"
end
```

When are Objects Saved?

Assigning an object to a `belongs_to` association does not automatically save the object. It does not save the associated object either.

4.4.2 `has_one` Association Reference

The `has_one` association creates a one-to-one match with another model. In database terms, this association says that the other class contains the foreign key. If this class contains the foreign key, then you should use `belongs_to` instead.

Methods Added by `has_one`

When you declare a `has_one` association, the declaring class automatically gains four methods related to the association:

- `association(force_reload = false)`

- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`

In all of these methods, `association` is replaced with the symbol passed as the first argument to `has_one`. For example, given the declaration:

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

Each instance of the `Supplier` model will have these methods:

```
account
account=
build_account
create_account
```

When initializing a new `has_one` or `belongs_to` association you must use the `build_` prefix to build the association, rather than the `association.build` method that would be used for `has_many` or `has_and_belongs_to_many` associations. To create one, use the `create_` prefix.

4.2.1.1 `association(force_reload = false)`

The `association` method returns the associated object, if any. If no associated object is found, it returns `nil`.

```
@account = @supplier.account
```

If the associated object has already been retrieved from the database for this object, the cached version will be returned. To override this behavior (and force a database read), pass `true` as the `force_reload` argument.

4.2.1.2 `association=(associate)`

The `association=` method assigns an associated object to this object. Behind the scenes, this means extracting the primary key from this object and setting the associate object's foreign key to the same value.

```
@supplier.account = @account
```

4.2.1.3 `build_association(attributes = {})`

The `build_association` method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through its foreign key will be set, but the associated object will not yet be saved.

```
@account = @supplier.build_account(:terms => "Net 30")
```

4.2.1.4 `create_association(attributes = {})`

The `create_association` method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through its foreign key will be set, and, once it passes all of the validations specified on the associated model, the associated object will be saved.

```
@account = @supplier.create_account(:terms => "Net 30")
```

Options for `has_one`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the `has_one` association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this assocation uses two such options:

```
class Supplier < ActiveRecord::Base
  has_one :account, :class_name => "Billing",
          :dependent => :nullify
end
```

The `has_one` association supports these options:

- `:as`
- `:autosave`
- `:class_name`
- `:conditions`
- `:dependent`
- `:foreign_key`
- `:include`
- `:inverse_of`
- `:order`
- `:primary_key`
- `:readonly`
- `:select`
- `:source`
- `:source_type`
- `:through`
- `:validate`

4.2.2.1 :as

Setting the **:as** option indicates that this is a polymorphic association. Polymorphic associations were discussed in detail earlier in this guide.

4.2.2.2 :autosave

If you set the **:autosave** option to **true**, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.2.2.3 :class_name

If the name of the other model cannot be derived from the association name, you can use the **:class_name** option to supply the model name. For example, if a supplier has an account, but the actual name of the model containing accounts is **Billing**, you'd set things up this way:

```
class Supplier < ActiveRecord::Base
  has_one :account, :class_name => "Billing"
end
```

4.2.2.4 :conditions

The **:conditions** option lets you specify the conditions that the associated object must meet (in the syntax used by an SQL**WHERE** clause).

```
class Supplier < ActiveRecord::Base
  has_one :account, :conditions => "confirmed = 1"
end
```

4.2.2.5 `:dependent`

If you set the **`:dependent`** option to **`:destroy`**, then deleting this object will call the **`destroy`** method on the associated object to delete that object. If you set the **`:dependent`** option to **`:delete`**, then deleting this object will delete the associated object without calling its **`destroy`** method. If you set the **`:dependent`** option to **`:nullify`**, then deleting this object will set the foreign key in the association object to `NULL`.

4.2.2.6 `:foreign_key`

By convention, Rails assumes that the column used to hold the foreign key on the other model is the name of this model with the suffix **`_id`** added. The **`:foreign_key`** option lets you set the name of the foreign key directly:

```
class Supplier < ActiveRecord::Base
  has_one :account, :foreign_key => "supp_id"
end
```

In any case, Rails will not create foreign key columns for you. You need to explicitly define them as part of your migrations.

4.2.2.7 `:include`

You can use the **`:include`** option to specify second-order associations that should be eager-loaded when this association is used. For example, consider these models:

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

If you frequently retrieve representatives directly from suppliers (`@supplier.account.representative`), then you can make your code somewhat more efficient by including representatives in the association from suppliers to accounts:

```
class Supplier < ActiveRecord::Base
  has_one :account, :include => :representative
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

4.2.2.8 `:inverse_of`

The `:inverse_of` option specifies the name of the `belongs_to` association that is the inverse of this association. Does not work in combination with the `:through` or `:as` options.

```
class Supplier < ActiveRecord::Base
  has_one :account, :inverse_of => :supplier
end

class Account < ActiveRecord::Base
  belongs_to :supplier, :inverse_of => :account
end
```

4.2.2.9 `:order`

The `:order` option dictates the order in which associated objects will be received (in the syntax used by an SQL`ORDER BY` clause). Because a `has_one` association will only retrieve a single associated object, this option should not be needed.

4.2.2.10 `:primary_key`

By convention, Rails assumes that the column used to hold the primary key of this model is `id`. You can override this and explicitly specify the primary key with the `:primary_key` option.

4.2.2.11 `:readonly`

If you set the `:readonly` option to `true`, then the associated object will be read-only when retrieved via the association.

4.2.2.12 `:select`

The `:select` option lets you override the SQL`SELECT` clause that is used to retrieve data about the associated object. By default, Rails retrieves all columns.

4.2.2.13 `:source`

The `:source` option specifies the source association name for a `has_one :through` association.

4.2.2.14 `:source_type`

The **`:source_type`** option specifies the source association type for a **`has_one :through`** association that proceeds through a polymorphic association.

4.2.2.15 `:through`

The **`:through`** option specifies a join model through which to perform the query. **`has_one :through`** associations were discussed in detail earlier in this guide.

4.2.2.16 `:validate`

If you set the **`:validate`** option to **`true`**, then associated objects will be validated whenever you save this object. By default, this is **`false`**: associated objects will not be validated when this object is saved.

Do Any Associated Objects Exist?

You can see if any associated objects exist by using the association.nil? method:

```
if @supplier.account.nil?
  @msg = "No account found for this supplier"
end
```

When are Objects Saved?

When you assign an object to a **`has_one`** association, that object is automatically saved (in order to update its foreign key). In addition, any object being replaced is also automatically saved, because its foreign key will change too.

If either of these saves fails due to validation errors, then the assignment statement returns `false` and the assignment itself is cancelled.

If the parent object (the one declaring the `has_one` association) is unsaved (that is, `new_record?` returns `true`) then the child objects are not saved. They will automatically when the parent object is saved.

If you want to assign an object to a `has_one` association without saving the object, use the `association.build` method.

4.4.3 `has_many` Association Reference

The `has_many` association creates a one-to-many relationship with another model. In database terms, this association says that the other class will have a foreign key that refers to instances of this class.

Methods Added by `has_many`

When you declare a `has_many` association, the declaring class automatically gains 13 methods related to the association:

- `collection(force_reload = false)`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection=objects`
- `collection_singular_ids`
- `collection_singular_ids=ids`
- `collection.clear`
- `collection.empty?`
- `collection.size`

- `collection.find(...)`
- `collection.where(...)`
- `collection.exists?(...)`
- `collection.build(attributes = {}, ...)`
- `collection.create(attributes = {})`

In all of these methods, `collection` is replaced with the symbol passed as the first argument to `has_many`, and `collection_singular` is replaced with the singularized version of that symbol.. For example, given the declaration:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Each instance of the customer model will have these methods:

```
orders(force_reload = false)
orders<<(object, ...)
orders.delete(object, ...)
orders=objects
order_ids
order_ids=ids
orders.clear
orders.empty?
orders.size
orders.find(...)
orders.where(...)
orders.exists?(...)
orders.build(attributes = {}, ...)
orders.create(attributes = {})
```

4.3.1.1 `collection(force_reload = false)`

The `collection` method returns an array of all of the associated objects. If there are no associated objects, it returns an empty array.

```
@orders = @customer.orders
```

4.3.1.2 `collection<<(object, ...)`

The `collection<<` method adds one or more objects to the collection by setting their foreign keys to the primary key of the calling model.

```
@customer.orders << @order1
```

4.3.1.3 `collection.delete(object, ...)`

The `collection.delete` method removes one or more objects from the collection by setting their foreign keys to `NULL`.

```
@customer.orders.delete(@order1)
```

Additionally, objects will be destroyed if they're associated with `:dependent => :destroy`, and deleted if they're associated with `:dependent => :delete_all`.

4.3.1.4 `collection=objects`

The `collection=` method makes the collection contain only the supplied objects, by adding and deleting as appropriate.

4.3.1.5 `collection_singular_ids`

The `collection_singular_ids` method returns an array of the ids of the objects in the collection.

```
@order_ids = @customer.order_ids
```

4.3.1.6 `collection_singular_ids=ids`

The **`collection_singular_ids=`** method makes the collection contain only the objects identified by the supplied primary key values, by adding and deleting as appropriate.

4.3.1.7 `collection.clear`

The **`collection.clear`** method removes every object from the collection. This destroys the associated objects if they are associated with **`:dependent => :destroy`**, deletes them directly from the database if **`:dependent => :delete_all`**, and otherwise sets their foreign keys to `NULL`.

4.3.1.8 `collection.empty?`

The **`collection.empty?`** method returns **`true`** if the collection does not contain any associated objects.

```
<% if @customer.orders.empty? %>
  No Orders Found
<% end %>
```

4.3.1.9 `collection.size`

The **`collection.size`** method returns the number of objects in the collection.

```
@order_count = @customer.orders.size
```

4.3.1.10 `collection.find(...)`

The `collection.find` method finds objects within the collection. It uses the same syntax and options as `ActiveRecord::Base.find`.

```
@open_orders = @customer.orders.where(:open => 1)
```

4.3.1.11 `collection.where(...)`

The `collection.where` method finds objects within the collection based on the conditions supplied but the objects are loaded lazily meaning that the database is queried only when the object(s) are accessed.

```
@open_orders = @customer.orders.where(:open => true)
# No query yet
@open_order = @open_orders.first
# Now the database will be queried
```

4.3.1.12 `collection.exists?(...)`

The `collection.exists?` method checks whether an object meeting the supplied conditions exists in the collection. It uses the same syntax and options as `ActiveRecord::Base.exists?`.

4.3.1.13 `collection.build(attributes = {}, ...)`

The `collection.build` method returns one or more new objects of the associated type. These objects will be instantiated from the passed attributes, and the link through their foreign key will be created, but the associated objects will not yet be saved.

```
@order = @customer.orders.build(:order_date => Time.now,
  :order_number => "A12345")
```

4.3.1.14 `collection.create(attributes = {})`

The `collection.create` method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through its foreign key will be created, and, once it passes all of the validations specified on the associated model, the associated object will be saved.

```
@order = @customer.orders.create(:order_date => Time.now,
  :order_number => "A12345")
```

Options for `has_many`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the `has_many` association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this assocation uses two such options:

```
class Customer < ActiveRecord::Base
has_many :orders, :dependent => :delete_all, :validate => :false
end
```

The `has_many` association supports these options:

- `:as`
- `:autosave`
- `:class_name`
- `:conditions`
- `:counter_sql`
- `:dependent`

- `:extend`
- `:finder_sql`
- `:foreign_key`
- `:group`
- `:include`
- `:inverse_of`
- `:limit`
- `:offset`
- `:order`
- `:primary_key`
- `:readonly`
- `:select`
- `:source`
- `:source_type`
- `:through`
- `:uniq`
- `:validate`

4.3.2.1 `:as`

Setting the `:as` option indicates that this is a polymorphic association, as discussed earlier in this guide.

4.3.2.2 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.3.2.3 :class_name

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if a customer has many orders, but the actual name of the model containing orders is `Transaction`, you'd set things up this way:

```
class Customer < ActiveRecord::Base
  has_many :orders, :class_name => "Transaction"
end
```

4.3.2.4 :conditions

The `:conditions` option lets you specify the conditions that the associated object must meet (in the syntax used by an SQL`WHERE` clause).

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, :class_name => "Order",
    :conditions => "confirmed = 1"
end
```

You can also set conditions via a hash:

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, :class_name => "Order",
    :conditions => { :confirmed => true }
end
```

If you use a hash-style `:conditions` option, then record creation via this association will be automatically scoped using the hash. In this case, using `@customer.confirmed_orders.create` or `@customer.confirmed_orders.build` will create orders where the confirmed column has the value `true`.

If you need to evaluate conditions dynamically at runtime, use a proc:

```
class Customer < ActiveRecord::Base
  has_many :latest_orders, :class_name => "Order",
  :conditions => proc { ["orders.created_at > ?", 10.hours.ago] }
end
```

4.3.2.5 `:counter_sql`

Normally Rails automatically generates the proper SQL to count the association members. With the `:counter_sql` option, you can specify a complete SQL statement to count them yourself.

If you specify `:finder_sql` but not `:counter_sql`, then the counter SQL will be generated by substituting `SELECT COUNT(*) FROM` for the `SELECT ... FROM` clause of your `:finder_sql` statement.

4.3.2.6 `:dependent`

If you set the `:dependent` option to `:destroy`, then deleting this object will call the `destroy` method on the associated objects to delete those objects. If you set the `:dependent` option to `:delete_all`, then deleting this object will delete the associated objects without calling their `destroy` method. If you set the `:dependent` option to `:nullify`, then deleting this object will set the foreign key in the associated objects to `NULL`.

This option is ignored when you use the `:through` option on the association.

4.3.2.7 `:extend`

The `:extend` option specifies a named module to extend the association proxy. Association extensions are discussed in detail later in this guide.

4.3.2.8 `:finder_sql`

Normally Rails automatically generates the proper SQL to fetch the association members. With the `:finder_sql` option, you can specify a complete SQL statement to fetch them yourself. If fetching objects requires complex multi-table SQL, this may be necessary.

4.3.2.9 `:foreign_key`

By convention, Rails assumes that the column used to hold the foreign key on the other model is the name of this model with the suffix `_id` added. The `:foreign_key` option lets you set the name of the foreign key directly:

```
class Customer < ActiveRecord::Base
  has_many :orders, :foreign_key => "cust_id"
end
```

In any case, Rails will not create foreign key columns for you. You need to explicitly define them as part of your migrations.

4.3.2.10 `:group`

The `:group` option supplies an attribute name to group the result set by, using a `GROUP BY` clause in the finder SQL.

```
class Customer < ActiveRecord::Base
has_many :line_items, :through => :orders, :group => "orders.id"
end
```

4.3.2.11 `:include`

You can use the `:include` option to specify second-order associations that should be eager-loaded when this association is used. For example, consider these models:

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

If you frequently retrieve line items directly from customers (`@customer.orders.line_items`), then you can make your code somewhat more efficient by including line items in the association from customers to orders:

```
class Customer < ActiveRecord::Base
  has_many :orders, :include => :line_items
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

4.3.2.12 `:inverse_of`

The `:inverse_of` option specifies the name of the `belongs_to` association that is the inverse of this association. Does not work in combination with the `:through` or `:as` options.

```
class Customer < ActiveRecord::Base
  has_many :orders, :inverse_of => :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, :inverse_of => :orders
end
```

4.3.2.13 `:limit`

The `:limit` option lets you restrict the total number of objects that will be fetched through an association.

```
class Customer < ActiveRecord::Base
  has_many :recent_orders, :class_name => "Order",
    :order => "order_date DESC", :limit => 100
end
```

4.3.2.14 `:offset`

The `:offset` option lets you specify the starting offset for fetching objects via an association. For example, if you set `:offset => 11`, it will skip the first 11 records.

4.3.2.15 `:order`

The `:order` option dictates the order in which associated objects will be received (in the syntax used by an SQL`ORDER BY` clause).

```
class Customer < ActiveRecord::Base
  has_many :orders, :order => "date_confirmed DESC"
end
```

4.3.2.16 `:primary_key`

By convention, Rails assumes that the column used to hold the primary key of the association is `id`. You can override this and explicitly specify the primary key with the **`:primary_key`** option.

4.3.2.17 `:readonly`

If you set the **`:readonly`** option to **`true`**, then the associated objects will be read-only when retrieved via the association.

4.3.2.18 `:select`

The **`:select`** option lets you override the SQL`SELECT` clause that is used to retrieve data about the associated objects. By default, Rails retrieves all columns.

If you specify your own **`:select`**, be sure to include the primary key and foreign key columns of the associated model. If you do not, Rails will throw an error.

4.3.2.19 `:source`

The **`:source`** option specifies the source association name for a **`has_many :through`** association. You only need to use this option if the name of the source association cannot be automatically inferred from the association name.

4.3.2.20 `:source_type`

The **`:source_type`** option specifies the source association type for a **`has_many :through`** association that proceeds through a polymorphic association.

4.3.2.21 `:through`

The **`:through`** option specifies a join model through which to perform the query. **`has_many :through`** associations provide a way to implement many-to-many relationships, as discussed earlier in this guide.

4.3.2.22 :uniq

Set the `:uniq` option to true to keep the collection free of duplicates. This is mostly useful together with the `:through` option.

```
class Person < ActiveRecord::Base
  has_many :readings
  has_many :posts, :through => :readings
end

person = Person.create(:name => 'john')
post   = Post.create(:name => 'a1')
person.posts << post
person.posts << post
person.posts.inspect
# => [#<Post id: 5, name: "a1">, #<Post id: 5, name: "a1">]
Reading.all.inspect
# => [#<Reading id: 12, person_id: 5, post_id: 5>,
      #<Reading id: 13, person_id: 5, post_id: 5>]
```

In the above case there are two readings and `person.posts` brings out both of them even though these records are pointing to the same post.

Now let's set `:uniq` to true:

```
class Person
  has_many :readings
  has_many :posts, :through => :readings, :uniq => true
end

person = Person.create(:name => 'honda')
post   = Post.create(:name => 'a1')
person.posts << post
person.posts << post
person.posts.inspect
# => [#<Post id: 7, name: "a1">]
Reading.all.inspect
# => [#<Reading id: 16, person_id: 7, post_id: 7>,
      #<Reading id: 17, person_id: 7, post_id: 7>]
```

In the above case there are still two readings. However `person.posts` shows only one post because the collection loads only unique records.

4.3.2.23 `:validate`

If you set the `:validate` option to `false`, then associated objects will not be validated whenever you save this object. By default, this is `true`: associated objects will be validated when this object is saved.

When are Objects Saved?

When you assign an object to a `has_many` association, that object is automatically saved (in order to update its foreign key). If you assign multiple objects in one statement, then they are all saved.

If any of these saves fails due to validation errors, then the assignment statement returns `false` and the assignment itself is cancelled.

If the parent object (the one declaring the `has_many` association) is unsaved (that is, `new_record?` returns `true`) then the child objects are not saved when they are added. All unsaved members of the association will automatically be saved when the parent is saved.

If you want to assign an object to a `has_many` association without saving the object, use the `collection.build` method.

4.4.4 `has_and_belongs_to_many` Association Reference

The `has_and_belongs_to_many` association creates a many-to-many relationship with another model. In database terms, this associates two classes via an intermediate join table that includes foreign keys referring to each of the classes.

Methods Added by `has_and_belongs_to_many`

When you declare a `has_and_belongs_to_many` association, the declaring class automatically gains 13 methods related to the association:

- `collection(force_reload = false)`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection=objects`
- `collection_singular_ids`
- `collection_singular_ids=ids`
- `collection.clear`
- `collection.empty?`
- `collection.size`
- `collection.find(...)`
- `collection.where(...)`
- `collection.exists?(...)`
- `collection.build(attributes = {})`
- `collection.create(attributes = {})`

In all of these methods, `collection` is replaced with the symbol passed as the first argument to `has_and_belongs_to_many`, and `collection_singular` is replaced with the singularized version of that symbol. For example, given the declaration:

```
class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Each instance of the part model will have these methods:

```
assemblies(force_reload = false)
assemblies<<(object, ...)
assemblies.delete(object, ...)
assemblies=objects
assembly_ids
assembly_ids=ids
assemblies.clear
assemblies.empty?
assemblies.size
assemblies.find(...)
assemblies.where(...)
assemblies.exists?(...)
assemblies.build(attributes = {}, ...)
assemblies.create(attributes = {})
```

4.4.1.1 Additional Column Methods

If the join table for a `has_and_belongs_to_many` association has additional columns beyond the two foreign keys, these columns will be added as attributes to records retrieved via that association. Records returned with additional attributes will always be read-only, because Rails cannot save changes to those attributes.

The use of extra attributes on the join table in a has_and_belongs_to_many association is deprecated. If you require this sort of complex behavior on the table that joins two models in a many-to-many relationship, you should use a **has_many :through** association instead of **has_and_belongs_to_many**.

4.4.1.2 collection(force_reload = false)

The **collection** method returns an array of all of the associated objects. If there are no associated objects, it returns an empty array.

```
@assemblies = @part.assemblies
```

4.4.1.3 collection<<(object, ...)

The **collection**<< method adds one or more objects to the collection by creating records in the join table.

```
@part.assemblies << @assembly1
```

This method is aliased as collection.concat and collection.push.

4.4.1.4 collection.delete(object, ...)

The **collection.delete** method removes one or more objects from the collection by deleting records in the join table. This does not destroy the objects.

```
@part.assemblies.delete(@assembly1)
```

4.4.1.5 collection=objects

The **collection=** method makes the collection contain only the supplied objects, by adding and deleting as appropriate.

4.4.1.6 `collection_singular_ids`

The `collection_singular_ids` method returns an array of the ids of the objects in the collection.

```
@assembly_ids = @part.assembly_ids
```

4.4.1.7 `collection_singular_ids=ids`

The `collection_singular_ids=` method makes the collection contain only the objects identified by the supplied primary key values, by adding and deleting as appropriate.

4.4.1.8 `collection.clear`

The `collection.clear` method removes every object from the collection by deleting the rows from the joining table. This does not destroy the associated objects.

4.4.1.9 `collection.empty?`

The `collection.empty?` method returns `true` if the collection does not contain any associated objects.

```
<% if @part.assemblies.empty? %>
  This part is not used in any assemblies
<% end %>
```

4.4.1.10 `collection.size`

The `collection.size` method returns the number of objects in the collection.

```
@assembly_count = @part.assemblies.size
```

4.4.1.11 `collection.find(...)`

The `collection.find` method finds objects within the collection. It uses the same syntax and options as `ActiveRecord::Base.find`. It also adds the additional condition that the object must be in the collection.

```
@new_assemblies = @part.assemblies.all(
  :conditions => ["created_at > ?", 2.days.ago])
```

Beginning with Rails 3, supplying options to the ActiveRecord::Base.find method is discouraged. Use collection.where instead when you need to pass conditions.

4.4.1.12 `collection.where(...)`

The `collection.where` method finds objects within the collection based on the conditions supplied but the objects are loaded lazily meaning that the database is queried only when the object(s) are accessed. It also adds the additional condition that the object must be in the collection.

```
@new_assemblies = @part.assemblies.where("created_at > ?", 2.days.ago)
```

4.4.1.13 `collection.exists?(...)`

The `collection.exists?` method checks whether an object meeting the supplied conditions exists in the collection. It uses the same syntax and options as `ActiveRecord::Base.exists?`.

4.4.1.14 `collection.build(attributes = {})`

The **`collection.build`** method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through the join table will be created, but the associated object will not yet be saved.

```
@assembly = @part.assemblies.build(
  {:assembly_name => "Transmission housing"})
```

4.4.1.15 `collection.create(attributes = {})`

The **`collection.create`** method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through the join table will be created, and, once it passes all of the validations specified on the associated model, the associated object will be saved.

```
@assembly = @part.assemblies.create(
  {:assembly_name => "Transmission housing"})
```

Options for `has_and_belongs_to_many`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the **`has_and_belongs_to_many`** association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this assocation uses two such options:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :uniq => true,
    :read_only => true
end
```

The `has_and_belongs_to_many` association supports these options:

- `:association_foreign_key`
- `:autosave`
- `:class_name`
- `:conditions`
- `:counter_sql`
- `:delete_sql`
- `:extend`
- `:finder_sql`
- `:foreign_key`
- `:group`
- `:include`
- `:insert_sql`
- `:join_table`
- `:limit`
- `:offset`
- `:order`
- `:readonly`
- `:select`
- `:uniq`
- `:validate`

4.4.2.1 `:association_foreign_key`

By convention, Rails assumes that the column in the join table used to hold the foreign key pointing to the other model is the name of that model with the suffix `_id` added. The `:association_foreign_key` option lets you set the name of the foreign key directly:

The `:foreign_key` and `:association_foreign_key` options are useful when setting up a many-to-many self-join. For example:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends, :class_name => "User",
    :foreign_key => "this_user_id",
    :association_foreign_key => "other_user_id"
end
```

4.4.2.2 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.4.2.3 `:class_name`

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if a part has many assemblies, but the actual name of the model containing assemblies is `Gadget`, you'd set things up this way:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :class_name => "Gadget"
end
```

4.4.2.4 `:conditions`

The `:conditions` option lets you specify the conditions that the associated object must meet (in the syntax used by an SQL`WHERE` clause).

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    :conditions => "factory = 'Seattle'"
end
```

You can also set conditions via a hash:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    :conditions => { :factory => 'Seattle' }
end
```

If you use a hash-style `:conditions` option, then record creation via this association will be automatically scoped using the hash. In this case, using `@parts.assemblies.create` or `@parts.assemblies.build` will create orders where the `factory` column has the value “Seattle”.

4.4.2.5 `:counter_sql`

Normally Rails automatically generates the proper SQL to count the association members. With the `:counter_sql` option, you can specify a complete SQL statement to count them yourself.

If you specify `:finder_sql` but not `:counter_sql`, then the counter SQL will be generated by substituting `SELECT COUNT(*) FROM` for the `SELECT ... FROM` clause of your `:finder_sql` statement.

4.4.2.6 `:delete_sql`

Normally Rails automatically generates the proper SQL to remove links between the associated classes. With the `:delete_sql` option, you can specify a complete SQL statement to delete them yourself.

4.4.2.7 `:extend`

The **`:extend`** option specifies a named module to extend the association proxy. Association extensions are discussed in detail later in this guide.

4.4.2.8 `:finder_sql`

Normally Rails automatically generates the proper SQL to fetch the association members. With the **`:finder_sql`** option, you can specify a complete SQL statement to fetch them yourself. If fetching objects requires complex multi-table SQL, this may be necessary.

4.4.2.9 `:foreign_key`

By convention, Rails assumes that the column in the join table used to hold the foreign key pointing to this model is the name of this model with the suffix **`_id`** added. The **`:foreign_key`** option lets you set the name of the foreign key directly:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends, :class_name => "User",
    :foreign_key => "this_user_id",
    :association_foreign_key => "other_user_id"
end
```

4.4.2.10 `:group`

The **`:group`** option supplies an attribute name to group the result set by, using a `GROUP BY` clause in the finder SQL.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :group => "factory"
end
```

4.4.2.11 `:include`

You can use the **`:include`** option to specify second-order associations that should be eager-loaded when this association is used.

4.4.2.12 `:insert_sql`

Normally Rails automatically generates the proper SQL to create links between the associated classes. With the **`:insert_sql`** option, you can specify a complete SQL statement to insert them yourself.

4.4.2.13 `:join_table`

If the default name of the join table, based on lexical ordering, is not what you want, you can use the **`:join_table`** option to override the default.

4.4.2.14 `:limit`

The **`:limit`** option lets you restrict the total number of objects that will be fetched through an association.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :order => "created_at DESC",
    :limit => 50
end
```

4.4.2.15 `:offset`

The **`:offset`** option lets you specify the starting offset for fetching objects via an association. For example, if you set **`:offset => 11`**, it will skip the first 11 records.

4.4.2.16 `:order`

The **`:order`** option dictates the order in which associated objects will be received (in the syntax used by an SQL`ORDER BY` clause).

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :order => "assembly_name ASC"
end
```

4.4.2.17 `:readonly`

If you set the `:readonly` option to `true`, then the associated objects will be read-only when retrieved via the association.

4.4.2.18 `:select`

The `:select` option lets you override the SQL`SELECT` clause that is used to retrieve data about the associated objects. By default, Rails retrieves all columns.

4.4.2.19 `:uniq`

Specify the `:uniq => true` option to remove duplicates from the collection.

4.4.2.20 `:validate`

If you set the `:validate` option to `false`, then associated objects will not be validated whenever you save this object. By default, this is `true`: associated objects will be validated when this object is saved.

When are Objects Saved?

When you assign an object to a `has_and_belongs_to_many` association, that object is automatically saved (in order to update the join table). If you assign multiple objects in one statement, then they are all saved.

If any of these saves fails due to validation errors, then the assignment statement returns `false` and the assignment itself is cancelled.

If the parent object (the one declaring the has_and_belongs_to_many association) is unsaved (that is, `new_record?` returns `true`) then the child objects are not saved when they are added. All unsaved members of the association will automatically be saved when the parent is saved.

If you want to assign an object to a `has_and_belongs_to_many` association without saving the object, use the `collection.build` method.

4.4.5 Association Callbacks

Normal callbacks hook into the life cycle of Active Record objects, allowing you to work with those objects at various points. For example, you can use a `:before_save` callback to cause something to happen just before an object is saved.

Association callbacks are similar to normal callbacks, but they are triggered by events in the life cycle of a collection. There are four available association callbacks:

- `before_add`
- `after_add`
- `before_remove`
- `after_remove`

You define association callbacks by adding options to the association declaration. For example:

```
class Customer < ActiveRecord::Base
  has_many :orders, :before_add => :check_credit_limit

  def check_credit_limit(order)
    ...
  end
end
```

Rails passes the object being added or removed to the callback.

You can stack callbacks on a single event by passing them as an array:

```
class Customer < ActiveRecord::Base
  has_many :orders,
    :before_add => [:check_credit_limit,
                    :calculate_shipping_charges]

  def check_credit_limit(order)
    ...
  end

  def calculate_shipping_charges(order)
    ...
  end
end
```

If a `before_add` callback throws an exception, the object does not get added to the collection. Similarly, if a `before_remove` callback throws an exception, the object does not get removed from the collection.

4.4.6 Association Extensions

You're not limited to the functionality that Rails automatically builds into association proxy objects. You can also extend these objects through anonymous modules, adding new finders, creators, or other methods. For example:

```
class Customer < ActiveRecord::Base
  has_many :orders do
    def find_by_order_prefix(order_number)
      find_by_region_id(order_number[0..2])
    end
  end
end
```

If you have an extension that should be shared by many associations, you can use a named extension module. For example:

```
module FindRecentExtension
  def find_recent
    where("created_at > ?", 5.days.ago)
  end
end

class Customer < ActiveRecord::Base
  has_many :orders, :extend => FindRecentExtension
end

class Supplier < ActiveRecord::Base
  has_many :deliveries, :extend => FindRecentExtension
end
```

To include more than one extension module in a single association, specify an array of modules:

```
class Customer < ActiveRecord::Base
  has_many :orders,
    :extend => [FindRecentExtension, FindActiveExtension]
end
```

Extensions can refer to the internals of the association proxy using these three attributes of the `proxy_association` accessor:

- `proxy_association.owner` returns the object that the association is a part of.
- `proxy_association.reflection` returns the reflection object

that describes the association.

- `proxy_association.target` returns the associated object for `belongs_to` or `has_one`, or the collection of associated objects for `has_many` or `has_and_belongs_to_many`.

Chapter 5

Active Record Query Interface

This guide covers different ways to retrieve data from the database using Active Record. By referring to this guide, you will be able to:

- Find records using a variety of methods and conditions
- Specify the order, retrieved attributes, grouping, and other properties of the found records
- Use eager loading to reduce the number of database queries needed for data retrieval
- Use dynamic finders methods
- Check for the existence of particular records
- Perform various calculations on Active Record models
- Run EXPLAIN on relations

This Guide is based on Rails 3.0. Some of the code shown here will not work in other versions of Rails.

If you're used to using raw SQL to find database records, then you will generally find that there are better ways to carry out the same operations in Rails. Active Record insulates you from the need to use SQL in most cases.

Code examples throughout this guide will refer to one or more of the following models:

All of the following models use id as the primary key, unless specified otherwise.

```
class Client < ActiveRecord::Base
  has_one :address
  has_many :orders
  has_and_belongs_to_many :roles
end

class Address < ActiveRecord::Base
  belongs_to :client
end

class Order < ActiveRecord::Base
  belongs_to :client, :counter_cache => true
end

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

Active Record will perform queries on the database for you and is compatible with most database systems (MySQL, PostgreSQL and SQLite to name a few). Regardless of which database system you're using, the Active Record method format will always be the same.

5.1 Retrieving Objects from the Database

To retrieve objects from the database, Active Record provides several finder methods. Each finder method allows you to pass arguments into it to perform certain queries on your database without writing raw SQL.

The methods are:

- where
- select
- group
- order
- reorder
- reverse_order
- limit
- offset
- joins
- includes
- lock
- readonly
- from
- having

All of the above methods return an instance of ActiveRecord::Relation.

The primary operation of Model.find(options) can be summarized as:

- Convert the supplied options to an equivalent SQL query.
- Fire the SQL query and retrieve the corresponding results from the database.
- Instantiate the equivalent Ruby object of the appropriate model for every resulting row.
- Run after_find callbacks, if any.

5.1.1 Retrieving a Single Object

Active Record provides five different ways of retrieving a single object.

Using a Primary Key

Using Model.find(primary_key), you can retrieve the object corresponding to the specified primary key that matches any supplied options. For example:

```
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

Model.find(primary_key) will raise an ActiveRecord::RecordNotFound exception if no matching record is found.

first

Model.first finds the first record matched by the supplied options, if any. For example:

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients LIMIT 1
```

Model.first returns nil if no matching record is found. No exception will be raised.

last

Model.last finds the last record matched by the supplied options. For example:

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

Model.last returns nil if no matching record is found. No exception will be raised.

first!

Model.first! finds the first record. For example:

```
client = Client.first!
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients LIMIT 1
```

Model.first! raises RecordNotFound if no matching record is found.

last!

Model.last! finds the last record. For example:

```
client = Client.last!
# => #<Client id: 221, first_name: "Russel">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

Model.last! raises RecordNotFound if no matching record is found.

5.1.2 Retrieving Multiple Objects

Using Multiple Primary Keys

Model.find(array_of_primary_key) accepts an array of primary keys, returning an array containing all of the matching records for the supplied primary keys. For example:

```
# Find the clients with primary keys 1 and 10.
client = Client.find([1, 10]) # Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">,
      #<Client id: 10, first_name: "Ryan">]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients WHERE (clients.id IN (1,10))
```

Model.find(array_of_primary_key) will raise an ActiveRecord::RecordNotFound exception unless a matching record is found for **all** of the supplied primary keys.

5.1.3 Retrieving Multiple Objects in Batches

We often need to iterate over a large set of records, as when we send a newsletter to a large set of users, or when we export data.

This may appear straightforward:

```
# This is very inefficient when the
# users table has thousands of rows.
User.all.each do |user|
  NewsLetter.weekly_deliver(user)
end
```

But this approach becomes increasingly impractical as the table size increases, since User.all.each instructs Active Record to fetch the entire table in a single pass, build a model object per row, and then keep the entire array of model objects in memory. Indeed, if we have a large number of records, the entire collection may exceed the amount of memory available.

Rails provides two methods that address this problem by dividing records into memory-friendly batches for processing. The first method, find_each, retrieves a batch of records and then yields each record to the block individually as a model. The second method, find_in_batches, retrieves a batch of records and then yields the entire batch to the block as an array of models.

The find_each and find_in_batches methods are intended for use in the batch processing of a large number of records that wouldn't fit in memory all at once. If you just need to loop over a thousand records the regular find methods are the preferred option.

find_each

The find_each method retrieves a batch of records and then yields each record to the block individually as a model. In the following example, find_each will retrieve 1000 records (the current default for both find_each and find_in_batches) and then yield each record individually to the block as a model. This process is repeated until all of the records have been processed:

```
User.find_each do |user|
  NewsLetter.weekly_deliver(user)
end
```

1.3.1.1 Options for find_each

The find_each method accepts most of the options allowed by the regular find method, except for :order and :limit, which are reserved for internal use by find_each.

Two additional options, :batch_size and :start, are available as well.

:batch_size

The :batch_size option allows you to specify the number of records to be retrieved in each batch, before being passed individually to the block. For example, to retrieve records in batches of 5000:

```
User.find_each(:batch_size => 5000) do |user|
  NewsLetter.weekly_deliver(user)
end
```

:start

By default, records are fetched in ascending order of the primary key, which must be an integer. The :start option allows you to configure the first ID of the sequence whenever the lowest ID is not the one you need. This would be useful, for example, if you wanted to resume an interrupted batch process, provided you saved the last processed ID as a checkpoint.

For example, to send newsletters only to users with the primary key starting from 2000, and to retrieve them in batches of 5000:

```
User.find_each(:start => 2000, :batch_size => 5000) do |user|
  NewsLetter.weekly_deliver(user)
end
```

Another example would be if you wanted multiple workers handling the same processing queue. You could have each worker handle 10000 records by setting the appropriate :start option on each worker.

The :include option allows you to name associations that should be loaded alongside with the models.

find_in_batches

The find_in_batches method is similar to find_each, since both retrieve batches of records. The difference is that find_in_batches yields batches to the block as an array of models, instead of individually. The following example will yield to the supplied block an array of up to 1000 invoices at a time, with the final block containing any remaining invoices:

```
# Give add_invoices an array of 1000 invoices at a time
Invoice.find_in_batches(:include => :invoice_lines) do |invoices|
  export.add_invoices(invoices)
end
```

The :include option allows you to name associations that should be loaded alongside with the models.

1.3.2.1 Options for find_in_batches

The find_in_batches method accepts the same :batch_size and :start options as find_each, as well as most of the options allowed by the regular find method, except for :order and :limit, which are reserved for internal use by find_in_batches.

5.2 Conditions

The where method allows you to specify conditions to limit the records returned, representing the WHERE-part of the SQL statement. Conditions can either be specified as a string, array, or hash.

5.2.1 Pure String Conditions

If you'd like to add conditions to your find, you could just specify them in there, just like Client.where("orders_count = '2'"). This will find all clients where the orders_count field's value is 2.

Building your own conditions as pure strings can leave you vulnerable to SQL injection exploits. For example, Client.where("first_name LIKE '%#{params[:first_name]

5.2.2 Array Conditions

Now what if that number could vary, say as an argument from somewhere? The find would then take the form:

```
Client.where("orders_count = ?", params[:orders])
```

Active Record will go through the first element in the conditions value and any additional elements will replace the question marks (?) in the first element.

If you want to specify multiple conditions:

```
Client.where("orders_count = ? AND locked = ?",
params[:orders], false)
```

In this example, the first question mark will be replaced with the value in params[:orders] and the second will be replaced with the SQL representation of false, which depends on the adapter.

This code is highly preferable:

```
Client.where("orders_count = ?", params[:orders])
```

to this code:

```
Client.where("orders_count = #{params[:orders]}")
```

because of argument safety. Putting the variable directly into the conditions string will pass the variable to the database **as-is**. This means that it will be an unescaped variable directly from a user who may have malicious intent. If you do this, you put your entire database at risk because once a user finds out he or she can exploit your database they can do just about anything to it. Never ever put your arguments directly inside the conditions string.

For more information on the dangers of SQL injection, see the Ruby on Rails Security Guide.

Placeholder Conditions

Similar to the (?) replacement style of params, you can also specify keys/values hash in your array conditions:

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
  {:start_date => params[:start_date], :end_date => params[:end_date]})
```

This makes for clearer readability if you have a large number of variable conditions.

Range Conditions

If you're looking for a range inside of a table (for example, users created in a certain timeframe) you can use the conditions option coupled with the INSQL statement for this. If you had two dates coming in from a controller you could do something like this to look for a range:

```
Client.where(:created_at =>
(params[:start_date].to_date)..(params[:end_date].to_date)
)
```

This query will generate something similar to the following SQL:

```
SELECT "clients".* FROM "clients" WHERE (
"clients"."created_at" BETWEEN '2010-09-29' AND '2010-11-30'
)
```

5.2.3 Hash Conditions

Active Record also allows you to pass in hash conditions which can increase the readability of your conditions syntax. With hash conditions, you pass in a hash with keys of the fields you want conditionalised and the values of how you want to conditionalise them:

Only equality, range and subset checking are possible with Hash conditions.

Equality Conditions

```
Client.where(:locked => true)
```

The field name can also be a string:

```
Client.where('locked' => true)
```

Range Conditions

The good thing about this is that we can pass in a range for our fields without it generating a large query as shown in the preamble of this section.

```
Client.where(:created_at =>
(Time.now.midnight - 1.day)..Time.now.midnight
)
```

This will find all clients created yesterday by using a BETWEEN-SQL statement:

```
SELECT * FROM clients WHERE (
clients.created_at BETWEEN '2008-12-21 00:00:00' AND '2008-12-22 00:00:00'
)
```

This demonstrates a shorter syntax for the examples in Array Conditions

Subset Conditions

If you want to find records using the IN expression you can pass an array to the conditions hash:

```
Client.where(:orders_count => [1,3,5])
```

This code will generate SQL like this:

```
SELECT * FROM clients WHERE (clients.orders_count IN (1,3,5))
```

5.3 Ordering

To retrieve records from the database in a specific order, you can use the order method.

For example, if you're getting a set of records and want to order them in ascending order by the created_at field in your table:

```
Client.order("created_at")
```

You could specify ASC or DESC as well:

```
Client.order("created_at DESC")
# OR
Client.order("created_at ASC")
```

Or ordering by multiple fields:

```
Client.order("orders_count ASC, created_at DESC")
```

5.4 Selecting Specific Fields

By default, Model.find selects all the fields from the result set using select *.

To select only a subset of fields from the result set, you can specify the subset via the select method.

If the select method is used, all the returning objects will be read only.

For example, to select only viewable_by and locked columns:

```
Client.select("viewable_by, locked")
```

The SQL query used by this find call will be somewhat like:

```
SELECT viewable_by, locked FROM clients
```

Be careful because this also means you're initializing a model object with only the fields that you've selected. If you attempt to access a field that is not in the initialized record you'll receive:

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

Where <attribute> is the attribute you asked for. The id method will not raise the ActiveRecord::MissingAttributeError, so just be careful when working with associations because they need the id method to function properly.

If you would like to only grab a single record per unique value in a certain field, you can use uniq:

```
Client.select(:name).uniq
```

This would generate SQL like:

```
SELECT DISTINCT name FROM clients
```

You can also remove the uniqueness constraint:

```
query = Client.select(:name).uniq
# => Returns unique names

query.uniq(false)
# => Returns all names, even if there are duplicates
```

5.5 Limit and Offset

To apply LIMIT to the SQL fired by the Model.find, you can specify the LIMIT using limit and offset methods on the relation.

You can use limit to specify the number of records to be retrieved, and use offset to specify the number of records to skip before starting to return the records. For example

```
Client.limit(5)
```

will return a maximum of 5 clients and because it specifies no offset it will return the first 5 in the table. The SQL it executes looks like this:

```
SELECT * FROM clients LIMIT 5
```

Adding offset to that

```
Client.limit(5).offset(30)
```

will return instead a maximum of 5 clients beginning with the 31st. The SQL looks like:

```
SELECT * FROM clients LIMIT 5 OFFSET 30
```

5.6 Group

To apply a GROUP BY clause to the SQL fired by the finder, you can specify the group method on the find.

For example, if you want to find a collection of the dates orders were created on:

```
Order.select("date(created_at) as ordered_date,
sum(price) as total_price").group("date(created_at)")
```

And this will give you a single Order object for each date where there are orders in the database.

The SQL that would be executed would be something like this:

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders GROUP BY date(created_at)
```

5.7 Having

SQL uses the HAVING clause to specify conditions on the GROUP BY fields. You can add the HAVING clause to the SQL fired by the Model.find by adding the :having option to the find.

For example:

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price")
.group("date(created_at)").having("sum(price) > ?", 100)
```

The SQL that would be executed would be something like this:

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders GROUP BY date(created_at) HAVING sum(price) > 100
```

This will return single order objects for each day, but only those that are ordered more than $100 in a day.

5.8 Overriding Conditions

5.8.1 except

You can specify certain conditions to be excepted by using the except method. For example:

```
Post.where('id > 10').limit(20).order('id asc').except(:order)
```

The SQL that would be executed:

```
SELECT * FROM posts WHERE id > 10 LIMIT 20
```

5.8.2 only

You can also override conditions using the only method. For example:

```
Post.where('id > 10').limit(20).order('id desc').only(:order, :where)
```

The SQL that would be executed:

```
SELECT * FROM posts WHERE id > 10 ORDER BY id DESC
```

5.8.3 reorder

The reorder method overrides the default scope order. For example:

```
class Post < ActiveRecord::Base
  ..
  ..
  has_many :comments, :order => 'posted_at DESC'
end

Post.find(10).comments.reorder('name')
```

The SQL that would be executed:

```
SELECT * FROM posts WHERE id = 10 ORDER BY name
```

In case the reorder clause is not used, the SQL executed would be:

```
SELECT * FROM posts WHERE id = 10 ORDER BY posted_at DESC
```

5.8.4 reverse_order

The reverse_order method reverses the ordering clause if specified.

```
Client.where("orders_count > 10").order(:name).reverse_order
```

The SQL that would be executed:

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY name DESC
```

If no ordering clause is specified in the query, the reverse_order orders by the primary key in reverse order.

```
Client.where("orders_count > 10").reverse_order
```

The SQL that would be executed:

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY clients.id DESC
```

This method accepts **no** arguments.

5.9 Readonly Objects

Active Record provides readonly method on a relation to explicitly disallow modification or deletion of any of the returned object. Any attempt to alter or destroy a readonly record will not succeed, raising an ActiveRecord::ReadOnlyRecord exception.

```
client = Client.readonly.first
client.visits += 1
client.save
```

As client is explicitly set to be a readonly object, the above code will raise an
ActiveRecord::ReadOnlyRecord exception when calling client.save with an updated value of visits.

5.10 Locking Records for Update

Locking is helpful for preventing race conditions when updating records in the database and ensuring atomic updates.

Active Record provides two locking mechanisms:

- Optimistic Locking
- Pessimistic Locking

5.10.1 Optimistic Locking

Optimistic locking allows multiple users to access the same record for edits, and assumes a minimum of conflicts with the data. It does this by checking whether another process has made changes to a record since it was opened. An ActiveRecord::StaleObjectError exception is thrown if that has occurred and the update is ignored.

Optimistic locking column

In order to use optimistic locking, the table needs to have a column called lock_version. Each time the record is updated, Active Record increments the lock_version column. If an update request is made with a lower value in the lock_version field than is currently in the lock_version column in the database, the update request will fail with an
ActiveRecord::StaleObjectError. Example:

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save
```

```
c2.name = "should fail"
c2.save # Raises an ActiveRecord::StaleObjectError
```

You're then responsible for dealing with the conflict by rescuing the exception and either rolling back, merging, or otherwise apply the business logic needed to resolve the conflict.

You must ensure that your database schema defaults the lock_version column to 0.

This behavior can be turned off by setting ActiveRecord::Base.lock_optimistically = false.

To override the name of the lock_version column, ActiveRecord::Base provides a class method called set_locking_column:

```
class Client < ActiveRecord::Base
  set_locking_column :lock_client_column
end
```

5.10.2 Pessimistic Locking

Pessimistic locking uses a locking mechanism provided by the underlying database. Using lock when building a relation obtains an exclusive lock on the selected rows. Relations using lock are usually wrapped inside a transaction for preventing deadlock conditions.

For example:

```
Item.transaction do
  i = Item.lock.first
  i.name = 'Jones'
  i.save
end
```

The above session produces the following SQL for a MySQL backend:

```
SQL (0.2ms)   BEGIN
Item Load (0.3ms)
SELECT * FROM `items` LIMIT 1 FOR UPDATE
```

```
Item Update(0.4ms)
UPDATE ‘items‘
SET ‘updated_at‘=’2009-02-07 18:05:56’, ‘name‘=’Jones’
WHERE ‘id‘=1
SQL (0.8ms)   COMMIT
```

You can also pass raw SQL to the lock method for allowing different types of locks. For example, MySQL has an expression called LOCK IN SHARE MODE where you can lock a record but still allow other queries to read it. To specify this expression just pass it in as the lock option:

```
Item.transaction do
  i = Item.lock("LOCK IN SHARE MODE").find(1)
  i.increment!(:views)
end
```

If you already have an instance of your model, you can start a transaction and acquire the lock in one go using the following code:

```
item = Item.first
item.with_lock do
  # This block is called within a transaction,
  # item is already locked.
  item.increment!(:views)
end
```

5.11 Joining Tables

Active Record provides a finder method called joins for specifying JOIN clauses on the resulting SQL. There are multiple ways to use the joins method.

5.11.1 Using a String SQL Fragment

You can just supply the raw SQL specifying the JOIN clause to joins:

```
Client.joins(
'LEFT OUTER JOIN addresses ON addresses.client_id = clients.id'
)
```

This will result in the following SQL:

```
SELECT clients.* FROM clients
LEFT OUTER JOIN addresses
ON addresses.client_id = clients.id
```

5.11.2 Using Array/Hash of Named Associations

This method only works with INNER JOIN.

Active Record lets you use the names of the associations defined on the model as a shortcut for specifying JOIN clause for those associations when using the joins method.

For example, consider the following Category, Post, Comments and Guest models:

```
class Category < ActiveRecord::Base
  has_many :posts
end

class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :post
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end

class Tag < ActiveRecord::Base
```

```
  belongs_to :post
end
```

Now all of the following will produce the expected join queries using INNER JOIN:

Joining a Single Association

```
Category.joins(:posts)
```

This produces:

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
```

Or, in English: "return a Category object for all categories with posts". Note that you will see duplicate categories if more than one post has the same category. If you want unique categories, you can use Category.joins(:post).select("distinct(categories.id)").

Joining Multiple Associations

```
Post.joins(:category, :comments)
```

This produces:

```
SELECT posts.* FROM posts
  INNER JOIN categories ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
```

Or, in English: "return all posts that have a category and at least one comment". Note again that posts with multiple comments will show up multiple times.

Joining Nested Associations (Single Level)

```
Post.joins(:comments => :guest)
```

This produces:

```
SELECT posts.* FROM posts
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
```

Or, in English: “return all posts that have a comment made by a guest.”

Joining Nested Associations (Multiple Level)

```
Category.joins(:posts => [{:comments => :guest}, :tags])
```

This produces:

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
  INNER JOIN tags ON tags.post_id = posts.id
```

5.11.3 Specifying Conditions on the Joined Tables

You can specify conditions on the joined tables using the regular Array and String conditions. Hash conditions provides a special syntax for specifying conditions for the joined tables:

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where('orders.created_at' => time_range)
```

An alternative and cleaner syntax is to nest the hash conditions:

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where(:orders => {:created_at => time_range})
```

This will find all clients who have orders that were created yesterday, again using a BETWEENSQL expression.

5.12 Eager Loading Associations

Eager loading is the mechanism for loading the associated records of the objects returned by Model.find using as few queries as possible.

N + 1 queries problem

Consider the following code, which finds 10 clients and prints their postcodes:

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

This code looks fine at the first sight. But the problem lies within the total number of queries executed. The above code executes 1 (to find 10 clients) + 10 (one per each client to load the address) = **11** queries in total.

Solution to N + 1 queries problem

Active Record lets you specify in advance all the associations that are going to be loaded. This is possible by specifying the includes method of the Model.find call. With includes, Active Record ensures that all of the specified associations are loaded using the minimum possible number of queries.

Revisiting the above case, we could rewrite Client.all to use eager load addresses:

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

The above code will execute just **2** queries, as opposed to **11** queries in the previous case:

```
SELECT * FROM clients LIMIT 10
SELECT addresses.* FROM addresses
  WHERE (addresses.client_id IN (1,2,3,4,5,6,7,8,9,10))
```

5.12.1 Eager Loading Multiple Associations

Active Record lets you eager load any number of associations with a single Model.find call by using an array, hash, or a nested hash of array/hash with the includes method.

Array of Multiple Associations

```
Post.includes(:category, :comments)
```

This loads all the posts and the associated category and comments for each post.

Nested Associations Hash

```
Category.includes(:posts => [{:comments => :guest}, :tags]).find(1)
```

This will find the category with id 1 and eager load all of the associated posts, the associated posts' tags and comments, and every comment's guest association.

5.12.2 Specifying Conditions on Eager Loaded Associations

Even though Active Record lets you specify conditions on the eager loaded associations just like joins, the recommended way is to use joins instead.

However if you must do this, you may use where as you would normally.

```
Post.includes(:comments).where("comments.visible", true)
```

This would generate a query which contains a LEFT OUTER JOIN whereas the joins method would generate one using the INNER JOIN function instead.

```
SELECT "posts"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5
FROM "posts"
LEFT OUTER JOIN "comments" ON "comments"."post_id" = "posts"."id"
WHERE (comments.visible = 1)
```

If there was no where condition, this would generate the normal set of two queries.

If, in the case of this includes query, there were no comments for any posts, all the posts would still be loaded. By using joins (an INNERJOIN), the join conditions **must** match, otherwise no records will be returned.

5.13 Scopes

Scoping allows you to specify commonly-used ARel queries which can be referenced as method calls on the association objects or models. With these scopes, you can use every method previously covered such as where, joins and includes. All scope methods will return an ActiveRecord::Relation object which will allow for further methods (such as other scopes) to be called on it.

To define a simple scope, we use the scope method inside the class, passing the ARel query that we'd like run when this scope is called:

```
class Post < ActiveRecord::Base
  scope :published, where(:published => true)
end
```

Just like before, these methods are also chainable:

```
class Post < ActiveRecord::Base
  scope :published, where(:published => true).joins(:category)
end
```

Scopes are also chainable within scopes:

```
class Post < ActiveRecord::Base
scope :published, where(:published => true)
scope :published_and_commented,
      published.and(self.arel_table[:comments_count].gt(0))
end
```

To call this published scope we can call it on either the class:

```
Post.published # => [published posts]
```

Or on an association consisting of Post objects:

```
category = Category.first
category.posts.published # => [published posts belonging to this category]
```

5.13.1 Working with times

If you're working with dates or times within scopes, due to how they are evaluated, you will need to use a lambda so that the scope is evaluated every time.

```
class Post < ActiveRecord::Base
  scope :last_week, lambda { where("created_at < ?", Time.zone.now ) }
end
```

Without the lambda, this Time.zone.now will only be called once.

5.13.2 Passing in arguments

When a lambda is used for a scope, it can take arguments:

```
class Post < ActiveRecord::Base
  scope :1_week_before, lambda { |time| where("created_at < ?", time) }
end
```

This may then be called using this:

```
Post.1_week_before(Time.zone.now)
```

However, this is just duplicating the functionality that would be provided to you by a class method.

```
class Post < ActiveRecord::Base
  def self.1_week_before(time)
    where("created_at < ?", time)
  end
end
```

Using a class method is the preferred way to accept arguments for scopes. These methods will still be accessible on the association objects:

```
category.posts.1_week_before(time)
```

5.13.3 Working with scopes

Where a relational object is required, the scoped method may come in handy. This will return an ActiveRecord::Relation object which can have further scoping applied to it afterwards. A place where this may come in handy is on associations

```
client = Client.find_by_first_name("Ryan")
orders = client.orders.scoped
```

With this new orders object, we are able to ascertain that this object can have more scopes applied to it. For instance, if we wanted to return orders only in the last 30 days at a later point.

```
orders.where("created_at > ?", 30.days.ago)
```

5.13.4 Applying a default scope

If we wish for a scope to be applied across all queries to the model we can use the default_scope method within the model itself.

```
class Client < ActiveRecord::Base
  default_scope where("removed_at IS NULL")
end
```

When queries are executed on this model, the SQL query will now look something like this:

```
SELECT * FROM clients WHERE removed_at IS NULL
```

5.13.5 Removing all scoping

If we wish to remove scoping for any reason we can use the unscoped method. This is especially useful if a default_scope is specified in the model and should not be applied for this particular query.

```
Client.unscoped.all
```

This method removes all scoping and will do a normal query on the table.

5.14 Dynamic Finders

For every field (also known as an attribute) you define in your table, Active Record provides a finder method. If you have a field called first_name on your Client model for example, you get find_by_first_name and find_all_by_first_name for free from Active Record. If you have a locked field on the Client model, you also get find_by_locked and find_all_by_locked methods.

You can also use find_last_by_* methods which will find the last record matching your argument.

You can specify an exclamation point (!) on the end of the dynamic finders to get them to raise an ActiveRecord::RecordNotFound error if they do not return any records, like Client.find_by_name!("Ryan")

If you want to find both by name and locked, you can chain these finders together by simply typing "and" between the fields. For example, Client.find_by_first_name_and_locked("Ryan", true).

Up to and including Rails 3.1, when the number of arguments passed to a dynamic finder method is lesser than the number of fields, say Client.find_by_name_and_locked("Ryan"), the behavior is to pass nil as the missing argument. This is **unintentional** and this behavior will be changed in Rails 3.2 to throw an ArgumentError.

5.15 Find or build a new object

It's common that you need to find a record or create it if it doesn't exist. You can do that with the first_or_create and first_or_create! methods.

5.15.1 first_or_create

The first_or_create method checks whether first returns nil or not. If it does return nil, then create is called. This is very powerful when coupled with the where method. Let's see an example.

Suppose you want to find a client named 'Andy', and if there's none, create one and additionally set his locked attribute to false. You can do so by running:

```
Client.where(:first_name => 'Andy').first_or_create(:locked => false)
#
 => #<Client id: 1, first_name: "Andy", orders_count: 0, locked:
false, created_at: "2011-08-30 06:09:27", updated_at: "2011-08-30
06:09:27">
```

The SQL generated by this method looks like this:

```
SELECT * FROM clients WHERE (clients.first_name = 'Andy') LIMIT 1
BEGIN
INSERT INTO clients
(created_at, first_name, locked, orders_count, updated_at)
VALUES ('2011-08-30 05:22:57', 'Andy', 0, NULL, '2011-08-30 05:22:57')
COMMIT
```

first_or_create returns either the record that already exists or the new record. In our case, we didn't already have a client named Andy so the record is created and returned.

The new record might not be saved to the database; that depends on whether validations passed or not (just like create).

It's also worth noting that first_or_create takes into account the arguments of the where method. In the example above we didn't explicitly pass a :first_name => 'Andy' argument to first_or_create. However, that was used when creating the new record because it was already passed before to the where method.

You can do the same with the find_or_create_by method:

```
Client.find_or_create_by_first_name(
:first_name => "Andy", :locked => false
)
```

This method still works, but it's encouraged to use first_or_create because it's more explicit on which arguments are used to find the record and which are used to create, resulting in less confusion overall.

5.15.2 first_or_create!

You can also use first_or_create! to raise an exception if the new record is invalid. Validations are not covered on this guide, but let's assume for a moment that you temporarily add

```
validates :orders_count, :presence => true
```

to your Client model. If you try to create a new Client without passing an orders_count, the record will be invalid and an exception will be raised:

```
Client.where(:first_name => 'Andy').first_or_create!(:locked => false)
# => ActiveRecord::RecordInvalid: Validation failed:
Orders count can't be blank
```

As with first_or_create there is a find_or_create_by! method but the first_or_create! method is preferred for clarity.

5.15.3 first_or_initialize

The first_or_initialize method will work just like first_or_create but it will not call create but new. This means that a new model instance will be created in memory but won't be saved to the database. Continuing with the first_or_create example, we now want the client named 'Nick':

```
nick = Client.where(:first_name => 'Nick')
      .first_or_initialize(:locked => false)
#
 => <Client id: nil, first_name: "Nick", orders_count: 0, locked:
false, created_at: "2011-08-30 06:09:27", updated_at: "2011-08-30
06:09:27">

nick.persisted?
# => false

nick.new_record?
# => true
```

Because the object is not yet stored in the database, the SQL generated looks like this:

```
SELECT * FROM clients WHERE (clients.first_name = 'Nick') LIMIT 1
```

When you want to save it to the database, just call save:

```
nick.save
# => true
```

5.16 Finding by SQL

If you'd like to use your own SQL to find records in a table you can use find_by_sql. The find_by_sql method will return an array of objects even if the underlying query returns just a single record. For example you could run this query:

```
Client.find_by_sql("SELECT * FROM clients
  INNER JOIN orders ON clients.id = orders.client_id
  ORDER clients.created_at desc")
```

find_by_sql provides you with a simple way of making custom calls to the database and retrieving instantiated objects.

5.17 select_all

find_by_sql has a close relative called connection#select_all. select_all will retrieve objects from the database using custom SQL just like find_by_sql but will not instantiate them. Instead, you will get an array of hashes where each hash indicates a record.

```
Client.connection.select_all("SELECT * FROM clients WHERE id = '1'")
```

5.18 pluck

pluck can be used to query a single column from the underlying table of a model. It accepts a column name as argument and returns an array of values of the specified column with the corresponding data type.

```
Client.where(:active => true).pluck(:id)
# SELECT id FROM clients WHERE active = 1

Client.uniq.pluck(:role)
# SELECT DISTINCT role FROM clients
```

pluck makes it possible to replace code like

```
Client.select(:id).map { |c| c.id }
```

with

```
Client.pluck(:id)
```

5.19 Existence of Objects

If you simply want to check for the existence of the object there's a method called exists?. This method will query the database using the same query as find, but instead of returning an object or collection of objects it will return either true or false.

```
Client.exists?(1
```

The exists? method also takes multiple ids, but the catch is that it will return true if any one of those records exists.

```
Client.exists?(1,2,3)
# or
Client.exists?([1,2,3])
```

It's even possible to use exists? without any arguments on a model or a relation.

```
Client.where(:first_name => 'Ryan').exists?
```

The above returns true if there is at least one client with the first_name 'Ryan' and false otherwise.

```
Client.exists?
```

The above returns false if the clients table is empty and true otherwise.

You can also use any? and many? to check for existence on a model or relation.

```
# via a model
Post.any?
Post.many?

# via a named scope
Post.recent.any?
Post.recent.many?

# via a relation
```

```
Post.where(:published => true).any?
Post.where(:published => true).many?

# via an association
Post.first.categories.any?
Post.first.categories.many?
```

5.20 Calculations

This section uses count as an example method in this preamble, but the options described apply to all sub-sections.

All calculation methods work directly on a model:

```
Client.count
# SELECT count(*) AS count_all FROM clients
```

Or on a relation:

```
Client.where(:first_name => 'Ryan').count
# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

You can also use various finder methods on a relation for performing complex calculations:

```
Client.includes("orders").where(
:first_name => 'Ryan', :orders => {:status => 'received'}
).count
```

Which will execute:

```
SELECT count(DISTINCT clients.id) AS count_all FROM clients
  LEFT OUTER JOIN orders ON orders.client_id = client.id WHERE
  (clients.first_name = 'Ryan' AND orders.status = 'received')
```

5.20.1 Count

If you want to see how many records are in your model's table you could call Client.count and that will return the number. If you want to be more specific and find all the clients with their age present in the database you can use Client.count(:age).

For options, please see the parent section, Calculations.

5.20.2 Average

If you want to see the average of a certain number in one of your tables you can call the average method on the class that relates to the table. This method call will look something like this:

```
Client.average("orders_count")
```

This will return a number (possibly a floating point number such as 3.14159265) representing the average value in the field.

For options, please see the parent section, Calculations.

5.20.3 Minimum

If you want to find the minimum value of a field in your table you can call the minimum method on the class that relates to the table. This method call will look something like this:

```
Client.minimum("age")
```

For options, please see the parent section, Calculations.

5.20.4 Maximum

If you want to find the maximum value of a field in your table you can call the maximum method on the class that relates to the table. This method call will look something like this:

```
Client.maximum("age")
```

For options, please see the parent section, Calculations.

5.20.5 Sum

If you want to find the sum of a field for all records in your table you can call the sum method on the class that relates to the table. This method call will look something like this:

```
Client.sum("orders_count")
```

For options, please see the parent section, Calculations.

5.21 Running EXPLAIN

You can run EXPLAIN on the queries triggered by relations. For example,

```
User.where(:id => 1).joins(:posts).explain
```

may yield

```
EXPLAIN for:
SELECT ‘users‘.* FROM ‘users‘
INNER JOIN ‘posts‘ ON ‘posts‘.‘user_id‘ = ‘users‘.‘id‘
WHERE ‘users‘.‘id‘ = 1

+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
|
 id | select_type | table | type  | possible_keys |
key     | key_len | ref   | rows |
Extra       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
|
 1 | SIMPLE      | users | const |
PRIMARY       | PRIMARY |
4       | const |    1
|
 |
|
 1 | SIMPLE      | posts | ALL   |
NULL          |
NULL    | NULL    | NULL
|    1 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
2 rows in set (0.00 sec)
```

under MySQL.

Active Record performs a pretty printing that emulates the one of the database shells. So, the same query running with the PostgreSQL adapter would yield instead

```
EXPLAIN for:
SELECT "users".* FROM "users"
INNER JOIN "posts" ON "posts"."user_id" = "users"."id"
WHERE "users"."id" = 1
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)
   Join Filter: (posts.user_id = users.id)
   ->  Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)
```

```
         Index Cond: (id = 1)
   ->  Seq Scan on posts  (cost=0.00..28.88 rows=8 width=4)
         Filter: (posts.user_id = 1)
(6 rows)
```

Eager loading may trigger more than one query under the hood, and some queries may need the results of previous ones. Because of that, explain actually executes the query, and then asks for the query plans. For example,

```
User.where(:id => 1).includes(:posts).explain
```

yields

```
EXPLAIN for: SELECT `users`.* FROM `users`  WHERE `users`.`id` = 1
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|
 id | select_type | table | type  | possible_keys |
key     | key_len | ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|
 1 | SIMPLE      | users | const |
PRIMARY       | PRIMARY |
4       | const |    1
|       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
1 row in set (0.00 sec)

EXPLAIN for: SELECT `posts`.* FROM `posts`  WHERE `posts`.`user_id` IN (1)
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|
 id | select_type | table | type | possible_keys | key  | key_len |
 ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|
 1 | SIMPLE      | posts | ALL  |
NULL          | NULL |
NULL    | NULL |    1 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

under MySQL.

5.21.1 Automatic EXPLAIN

Active Record is able to run EXPLAIN automatically on slow queries and log its output. This feature is controlled by the configuration parameter

```
config.active_record.auto_explain_threshold_in_seconds
```

If set to a number, any query exceeding those many seconds will have its EXPLAIN automatically triggered and logged. In the case of relations, the threshold is compared to the total time needed to fetch records. So, a relation is seen as a unit of work, no matter whether the implementation of eager loading involves several queries under the hood.

A threshold of nil disables automatic EXPLAINs.

The default threshold in development mode is 0.5 seconds, and nil in test and production modes.

Automatic EXPLAIN gets disabled if Active Record has no logger, regardless of the value of the threshold.

Disabling Automatic EXPLAIN

Automatic EXPLAIN can be selectively silenced with ActiveRecord::Base.silence_auto_explain:

```
ActiveRecord::Base.silence_auto_explain do
  # no automatic EXPLAIN is triggered here
end
```

That may be useful for queries you know are slow but fine, like a heavyweight report of an admin interface.

As its name suggests, silence_auto_explain only silences automatic EXPLAINs. Explicit calls to ActiveRecord::Relation#explain run.

5.21.2 Interpreting EXPLAIN

Interpretation of the output of EXPLAIN is beyond the scope of this guide. The following pointers may be helpful:

- SQLite3: EXPLAINQUERYPLAN

- MySQL: EXPLAIN Output Format
- PostgreSQL: Using EXPLAIN

Chapter 6

Layouts and Rendering in Rails

This guide covers the basic layout features of Action Controller and Action View. By referring to this guide, you will be able to:

- Use the various rendering methods built into Rails
- Create layouts with multiple content sections
- Use partials to DRY up your views
- Use nested layouts (sub-templates)

6.1 Overview: How the Pieces Fit Together

This guide focuses on the interaction between Controller and View in the Model-View-Controller triangle. As you know, the Controller is responsible for orchestrating the whole process of handling a request in Rails, though it normally hands off any heavy code to the Model. But then, when it's time to send a response back to the user, the Controller hands things off to the View. It's that handoff that is the subject of this guide.

In broad strokes, this involves deciding what should be sent as the response and calling an appropriate method to create that response. If the response is a full-blown view, Rails also does some extra work to wrap the view in a layout and possibly to pull in partial views. You'll see all of those paths later in this guide.

6.2 Creating Responses

From the controller's point of view, there are three ways to create an HTTP response:

- Call render to create a full response to send back to the browser
- Call redirect_to to send an HTTP redirect status code to the browser
- Call head to create a response consisting solely of HTTP headers to send back to the browser

I'll cover each of these methods in turn. But first, a few words about the very easiest thing that the controller can do to create a response: nothing at all.

6.2.1 Rendering by Default: Convention Over Configuration in Action

You've heard that Rails promotes "convention over configuration". Default rendering is an excellent example of this. By default, controllers in Rails automatically render views with names that correspond to valid routes. For example, if you have this code in your BooksController class:

```
class BooksController < ApplicationController
end
```

And the following in your routes file:

```
resources :books
```

And you have a view file app/views/books/index.html.erb:

```
<h1>Books are coming soon!</h1>
```

Rails will automatically render app/views/books/index.html.erb when you navigate to /books and you will see "Books are coming soon!" on your screen.

However a coming soon screen is only minimally useful, so you will soon create your Book model and add the index action to BooksController:

```
class BooksController < ApplicationController
  def index
    @books = Book.all
  end
end
```

Note that we don't have explicit render at the end of the index action in accordance with "convention over configuration" principle. The rule is that if you do not explicitly render something at the end of a controller action, Rails will automatically look for the action_name.html.erb template in the controller's view path and render it. So in this case, Rails will render the app/views/books/index.html.erb file.

If we want to display the properties of all the books in our view, we can do so with an ERB template like this:

```
<h1>Listing Books</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Summary</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @books.each do |book| %>
```

```
  <tr>
    <td><%= book.title %></td>
    <td><%= book.content %></td>
    <td><%= link_to 'Show', book %></td>
    <td><%= link_to 'Edit', edit_book_path(book) %></td>
    <td><%= link_to 'Remove', book, :confirm =>
        'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New book', new_book_path %>
```

The actual rendering is done by subclasses of ActionView::TemplateHandlers. This guide does not dig into that process, but it's important to know that the file extension on your view controls the choice of template handler. Beginning with Rails 2, the standard extensions are .erb for ERB (HTML with embedded Ruby), and .builder for Builder (XML generator).

6.2.2 Using render

In most cases, the ActionController::Base#render method does the heavy lifting of rendering your application's content for use by a browser. There are a variety of ways to customize the behaviour of render. You can render the default view for a Rails template, or a specific template, or a file, or inline code, or nothing at all. You can render text, JSON, or XML. You can specify the content type or HTTP status of the rendered response as well.

If you want to see the exact results of a call to render without needing to inspect it in a browser, you can call render_to_string. This method takes exactly the same options as render, but it returns a string instead of sending a response back to the browser.

Rendering Nothing

Perhaps the simplest thing you can do with render is to render nothing at all:

```
render :nothing => true
```

If you look at the response for this using cURL, you will see the following:

```
$ curl -i 127.0.0.1:3000/books
HTTP/1.1 200 OK
Connection: close
Date: Sun, 24 Jan 2010 09:25:18 GMT
Transfer-Encoding: chunked
Content-Type: */*; charset=utf-8
X-Runtime: 0.014297
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache

$
```

We see there is an empty response (no data after the Cache-Control line), but the request was successful because Rails has set the response to 200 OK. You can set the :status option on render to change this response. Rendering nothing can be useful for AJAX requests where all you want to send back to the browser is an acknowledgment that the request was completed.

You should probably be using the head method, discussed later in this guide, instead of render :nothing. This provides additional flexibility and makes it explicit that you're only generating HTTP headers.

Rendering an Action's View

If you want to render the view that corresponds to a different action within the same template, you can use render with the name of the view:

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to(@book)
  else
    render "edit"
  end
end
```

If the call to update_attributes fails, calling the update action in this controller will render the edit.html.erb template belonging to the same controller.

If you prefer, you can use a symbol instead of a string to specify the action to render:

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to(@book)
  else
    render :edit
  end
end
```

To be explicit, you can use render with the :action option (though this is no longer necessary in Rails 3.0):

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to(@book)
  else
    render :action => "edit"
  end
end
```

Using render with :action is a frequent source of confusion for Rails newcomers. The specified action is used to determine which view to render, but Rails does not run any of the code for that action in the controller. Any instance variables that you require in the view must be set up in the current action before calling render.

Rendering an Action's Template from Another Controller

What if you want to render a template from an entirely different controller from the one that contains the action code? You can also do that with render, which accepts the full path (relative to app/views) of the template to render. For example, if you're running code in an AdminProductsController that lives in app/controllers/admin, you can render the results of an action to a template in app/views/products this way:

```
render 'products/show'
```

Rails knows that this view belongs to a different controller because of the embedded slash character in the string. If you want to be explicit, you can use the :template option (which was required on Rails 2.2 and earlier):

```
render :template => 'products/show
```

Rendering an Arbitrary File

The render method can also use a view that's entirely outside of your application (perhaps you're sharing views between two Rails applications):

```
render "/u/apps/warehouse_app/current/app/views/products/show"
```

Rails determines that this is a file render because of the leading slash character. To be explicit, you can use the :file option (which was required on Rails 2.2 and earlier):

```
render :file =>
  "/u/apps/warehouse_app/current/app/views/products/show"
```

The :file option takes an absolute file-system path. Of course, you need to have rights to the view that you're using to render the content.

By default, the file is rendered without using the current layout. If you want Rails to put the file into the current layout, you need to add the :layout => true option.

If you're running Rails on Microsoft Windows, you should use the :file option to render a file, because Windows filenames do not have the same format as Unix filenames.

Wrapping it up

The above three ways of rendering (rendering another template within the controller, rendering a template within another controller and rendering an arbitrary file on the file system) are actually variants of the same action.

In fact, in the BooksController class, inside of the update action where we want to render the edit template if the book does not update successfully, all of the following render calls would all render the edit.html.erb template in the views/books directory:

```
render :edit
render :action => :edit
render 'edit'
render 'edit.html.erb'
render :action => 'edit'
render :action => 'edit.html.erb'
render 'books/edit'
render 'books/edit.html.erb'
render :template => 'books/edit'
render :template => 'books/edit.html.erb'
render '/path/to/rails/app/views/books/edit'
render '/path/to/rails/app/views/books/edit.html.erb'
render :file => '/path/to/rails/app/views/books/edit'
render :file => '/path/to/rails/app/views/books/edit.html.erb'
```

Which one you use is really a matter of style and convention, but the rule of thumb is to use the simplest one that makes sense for the code you are writing.

Using render with :inline

The render method can do without a view completely, if you're willing to use the :inline option to supply ERB as part of the method call. This is perfectly valid:

```
render :inline =>
  "<% products.each do |p| %><p><%= p.name %></p><% end %>"
```

There is seldom any good reason to use this option. Mixing ERB into your controllers defeats the MVC orientation of Rails and will make it harder for other developers to follow the logic of your project. Use a separate erb view instead.

By default, inline rendering uses ERB. You can force it to use Builder instead with the :type option:

```
render :inline =>
  "xml.p {'Horrid coding practice!'}", :type => :builder
```

Rendering Text

You can send plain text – with no markup at all – back to the browser by using the :text option to render:

```
render :text => "OK"
```

Rendering pure text is most useful when you're responding to AJAX or web service requests that are expecting something other than proper HTML.

By default, if you use the :text option, the text is rendered without using the current layout. If you want Rails to put the text into the current layout, you need to add the :layout => true option.

Rendering JSON

JSON is a JavaScript data format used by many AJAX libraries. Rails has built-in support for converting objects to JSON and rendering that JSON back to the browser:

```
render :json => @product
```

You don't need to call to_json on the object that you want to render. If you use the :json option, render will automatically call to_json for you.

Rendering XML

Rails also has built-in support for converting objects to XML and rendering that XML back to the caller:

```
render :xml => @product
```

You don't need to call to_xml on the object that you want to render. If you use the :xml option, render will automatically call to_xml for you.

Rendering Vanilla JavaScript

Rails can render vanilla JavaScript:

```
render :js => "alert('Hello Rails');"
```

This will send the supplied string to the browser with a MIME type of text/javascript.

Options for render

Calls to the render method generally accept four options:

- :content_type
- :layout
- :status
- :location

2.2.11.1 The :content_type Option

By default, Rails will serve the results of a rendering operation with the MIME content-type of text/html (or application/json if you use the :json option, or application/xml for the :xml option.). There are times when you might like to change this, and you can do so by setting the :content_type option:

```
render :file => filename, :content_type => 'application/rss'
```

2.2.11.2 The :layout Option

With most of the options to render, the rendered content is displayed as part of the current layout. You'll learn more about layouts and how to use them later in this guide.

You can use the :layout option to tell Rails to use a specific file as the layout for the current action:

```
render :layout => 'special_layout'
```

You can also tell Rails to render with no layout at all:

```
render :layout => false
```

2.2.11.3 The :status Option

Rails will automatically generate a response with the correct HTTP status code (in most cases, this is 200 OK). You can use the :status option to change this:

```
render :status => 500
render :status => :forbidden
```

Rails understands both numeric and symbolic status codes.

2.2.11.4 The :location Option

You can use the :location option to set the HTTPLocation header:

```
render :xml => photo, :location => photo_url(photo)
```

Finding Layouts

To find the current layout, Rails first looks for a file in app/views/layouts with the same base name as the controller. For example, rendering actions from the PhotosController class will use app/views/layouts/photos.html.erb (or app/views/layouts/photos.builder). If there is no such controller-specific layout, Rails will use app/views/layouts/application.html.erb or app/views/layouts/application.builder. If there is no .erb layout, Rails will use a .builder layout if one exists. Rails also provides several ways to more precisely assign specific layouts to individual controllers and actions.

2.2.12.1 Specifying Layouts for Controllers

You can override the default layout conventions in your controllers by using the layout declaration. For example:

```
class ProductsController < ApplicationController
  layout "inventory"
  #...
end
```

With this declaration, all of the methods within ProductsController will use app/views/layouts/inventory.html.erb for their layout.

To assign a specific layout for the entire application, use a layout declaration in your ApplicationController class:

```
class ApplicationController < ActionController::Base
  layout "main"
  #...
end
```

With this declaration, all of the views in the entire application will use app/views/layouts/main.html.erb for their layout.

2.2.12.2 Choosing Layouts at Runtime

You can use a symbol to defer the choice of layout until a request is processed:

```
class ProductsController < ApplicationController
  layout :products_layout

  def show
    @product = Product.find(params[:id])
  end

  private
    def products_layout
      @current_user.special? ? "special" : "products"
    end

end
```

Now, if the current user is a special user, they'll get a special layout when viewing a product.

You can even use an inline method, such as a Proc, to determine the layout. For example, if you pass a Proc object, the block you give the Proc will be given the controller instance, so the layout can be determined based on the current request:

```
class ProductsController < ApplicationController
layout Proc.new {
|controller| controller.request.xhr? ? 'popup' : 'application'
}
end
```

2.2.12.3 Conditional Layouts

Layouts specified at the controller level support the :only and :except options. These options take either a method name, or an array of method names, corresponding to method names within the controller:

```
class ProductsController < ApplicationController
  layout "product", :except => [:index, :rss]
end
```

With this declaration, the product layout would be used for everything but the rss and index methods.

2.2.12.4 Layout Inheritance

Layout declarations cascade downward in the hierarchy, and more specific layout declarations always override more general ones. For example:

- application_controller.rb

```
class ApplicationController < ActionController::Base
  layout "main"
end
```

- posts_controller.rb

```
class PostsController < ApplicationController
end
```

- special_posts_controller.rb

```
class SpecialPostsController < PostsController
  layout "special"
end
```

- old_posts_controller.rb

```
class OldPostsController < SpecialPostsController
  layout nil

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

  def index
    @old_posts = Post.older
    render :layout => "old"
  end
  # ...
end
```

In this application:

- In general, views will be rendered in the main layout
- PostsController#index will use the main layout
- SpecialPostsController#index will use the special layout
- OldPostsController#show will use no layout at all
- OldPostsController#index will use the old layout

Avoiding Double Render Errors

Sooner or later, most Rails developers will see the error message "Can only render or redirect once per action". While this is annoying, it's relatively easy to fix. Usually it happens because of a fundamental misunderstanding of the way that render works.

For example, here's some code that will trigger this error:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render :action => "special_show"
  end
  render :action => "regular_show"
end
```

If @book.special? evaluates to true, Rails will start the rendering process to dump the @book variable into the special_show view. But this will not stop the rest of the code in the show action from running, and when Rails hits the end of the action, it will start to render the regular_show view – and throw an error. The solution is simple: make sure that you have only one call to render or redirect in a single code path. One thing that can help is and return. Here's a patched version of the method:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render :action => "special_show" and return
  end
  render :action => "regular_show"
end
```

Make sure to use and return instead of && return because && return will not work due to the operator precedence in the Ruby Language.

Note that the implicit render done by ActionController detects if render has been called, so the following will work without errors:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render :action => "special_show"
  end
end
```

This will render a book with special? set with the special_show template, while other books will render with the default show template.

6.2.3 Using redirect_to

Another way to handle returning responses to an HTTP request is with redirect_to. As you've seen, render tells Rails which view (or other asset) to use in constructing a response. The redirect_to method does something completely different: it tells the browser to send a new request for a different URL. For example, you could redirect from wherever you are in your code to the index of photos in your application with this call:

```
redirect_to photos_url
```

You can use redirect_to with any arguments that you could use with link_to or url_for. There's also a special redirect that sends the user back to the page they just came from:

```
redirect_to :back
```

Getting a Different Redirect Status Code

Rails uses HTTP status code 302, a temporary redirect, when you call redirect_to. If you'd like to use a different status code, perhaps 301, a permanent redirect, you can use the :status option:

```
redirect_to photos_path, :status => 301
```

Just like the :status option for render, :status for redirect_to accepts both numeric and symbolic header designations.

The Difference Between render and redirect_to

Sometimes inexperienced developers think of redirect_to as a sort of goto command, moving execution from one place to another in your Rails code. This is <u>not</u> correct. Your code stops running and waits for a new request for the browser. It just happens that you've told the browser what request it should make next, by sending back an HTTP 302 status code.

Consider these actions to see the difference:

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by_id(params[:id])
  if @book.nil?
    render :action => "index"
  end
end
```

With the code in this form, there will likely be a problem if the @book variable is nil. Remember, a render :action doesn't run any code in the target action, so nothing will set up the @books variable that the index view will probably require. One way to fix this is to redirect instead of rendering:

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by_id(params[:id])
  if @book.nil?
    redirect_to :action => :index
  end
end
```

With this code, the browser will make a fresh request for the index page, the code in the index method will run, and all will be well.

The only downside to this code is that it requires a round trip to the browser: the browser requested the show action with /books/1 and the controller finds that there are no books, so the controller sends out a 302 redirect response to the browser telling it to go to /books/, the browser complies and sends a new request back to the controller asking now for the index action, the controller then gets all the books in the database and renders the index template, sending it back down to the browser which then shows it on your screen.

While in a small application, this added latency might not be a problem, it is something to think about if response time is a concern. We can demonstrate one way to handle this with a contrived example:

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by_id(params[:id])
  if @book.nil?
    @books = Book.all
    render "index", :alert => 'Your book was not found!'
  end
end
```

This would detect that there are no books with the specified ID, populate the @books instance variable with all the books in the model, and then directly render the index.html.erb template, returning it to the browser with a flash alert message to tell the user what happened.

6.2.4 Using head To Build Header-Only Responses

The head method can be used to send responses with only headers to the browser. It provides a more obvious alternative to calling render :nothing. The head method takes one parameter, which is interpreted as a hash of header names and values. For example, you can return only an error header:

```
head :bad_request
```

This would produce the following header:

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

Or you can use other HTTP headers to convey other information:

```
head :created, :location => photo_path(@photo)
```

Which would produce:

```
HTTP/1.1 201 Created
Connection: close
Date: Sun, 24 Jan 2010 12:16:44 GMT
Transfer-Encoding: chunked
Location: /photos/1
```

```
Content-Type: text/html; charset=utf-8
X-Runtime: 0.083496
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

6.3 Structuring Layouts

When Rails renders a view as a response, it does so by combining the view with the current layout, using the rules for finding the current layout that were covered earlier in this guide. Within a layout, you have access to three tools for combining different bits of output to form the overall response:

- Asset tags
- yield and content_for
- Partials

6.3.1 Asset Tag Helpers

Asset tag helpers provide methods for generating HTML that link views to feeds, JavaScript, stylesheets, images, videos and audios. There are six asset tag helpers available in Rails:

- auto_discovery_link_tag
- javascript_include_tag
- stylesheet_link_tag
- image_tag
- video_tag
- audio_tag

You can use these tags in layouts or other views, although the auto_discovery_link_tag, javascript_include_tag, and stylesheet_link_tag, are most commonly used in the <head> section of a layout.

The asset tag helpers do not verify the existence of the assets at the specified locations; they simply assume that you know what you're doing and generate the link.

Linking to Feeds with the auto_discovery_link_tag

The auto_discovery_link_tag helper builds HTML that most browsers and newsreaders can use to detect the presences of RSS or ATOM feeds. It takes the type of the link (:rss or :atom), a hash of options that are passed through to url_for, and a hash of options for the tag:

```
<%= auto_discovery_link_tag(:rss, {:action => "feed"},
  {:title => "RSS Feed"}) %>
```

There are three tag options available for the auto_discovery_link_tag:

- :rel specifies the rel value in the link. The default value is “alternate”.
- :type specifies an explicit MIME type. Rails will generate an appropriate MIME type automatically.
- :title specifies the title of the link. The default value is the upshifted :type value, for example, “ATOM” or “RSS”.

Linking to JavaScript Files with the javascript_include_tag

The javascript_include_tag helper returns an HTMLscript tag for each source provided.

If you are using Rails with the Asset Pipeline enabled, this helper will generate a link to /assets/javascripts/ rather than public/javascripts which was used in earlier versions of Rails. This link is then served by the Sprockets gem, which was introduced in Rails 3.1.

A JavaScript file within a Rails application or Rails engine goes in one of three locations: app/assets, lib/assets or vendor/assets.

These locations are explained in detail in the Asset Organization section in the Asset Pipeline Guide

You can specify a full path relative to the document root, or a URL, if you prefer. For example, to link to a JavaScript file that is inside a directory called javascripts inside of one of app/assets, lib/assets or vendor/assets, you would do this:

```
<%= javascript_include_tag "main" %>
```

Rails will then output a script tag such as this:

```
<script src='/assets/main.js' type="text/javascript"></script>
```

The request to this asset is then served by the Sprockets gem.

To include multiple files such as app/assets/javascripts/main.js and app/assets/javascripts/columns.js at the same time:

```
<%= javascript_include_tag "main", "columns" %>
```

To include app/assets/javascripts/main.js
and app/assets/javascripts/photos/columns.js:

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

To include http://example.com/main.js:

```
<%= javascript_include_tag "http://example.com/main.js" %>
```

If the application does not use the asset pipeline, the :defaults option loads jQuery by default:

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

Outputting script tags such as this:

```
<script src="/javascripts/jquery.js" type="text/javascript"></script>
<script src="/javascripts/jquery_ujs.js" type="text/javascript"></script>
```

These two files for jQuery, jquery.js and jquery_ujs.js must be placed inside public/javascripts if the application doesn't use the asset pipeline. These files can be downloaded from the jquery-rails repository on GitHub

If you are using the asset pipeline, this tag will render a script tag for an asset called defaults.js, which would not exist in your application unless you've explicitly defined it to be.

And you can in any case override the :defaults expansion in config/application.rb:

```
config.action_view.javascript_expansions[:defaults] = %w(foo.js bar.js)
```

You can also define new defaults:

```
config.action_view.javascript_expansions[:projects]
  = %w(projects.js tickets.js)
```

And use them by referencing them exactly like :defaults:

```
<%= javascript_include_tag :projects %>
```

When using :defaults, if an application.js file exists in public/javascripts it will be included as well at the end.

Also, if the asset pipeline is disabled, the :all expansion loads every JavaScript file in public/javascripts:

```
<%= javascript_include_tag :all %>
```

Note that your defaults of choice will be included first, so they will be available to all subsequently included files.

You can supply the :recursive option to load files in subfolders of public/javascripts as well:

```
<%= javascript_include_tag :all, :recursive => true %>
```

If you're loading multiple JavaScript files, you can create a better user experience by combining multiple files into a single download. To make this happen in production, specify :cache => true in your javascript_include_tag:

```
<%= javascript_include_tag "main", "columns", :cache => true %>
```

By default, the combined file will be delivered as javascripts/all.js. You can specify a location for the cached asset file instead:

```
<%= javascript_include_tag "main", "columns",
  :cache => 'cache/main/display' %>
```

You can even use dynamic paths such as cache/#{current_sitemain/display.

Linking to CSS Files with the stylesheet_link_tag

The stylesheet_link_tag helper returns an HTML<link> tag for each source provided.

If you are using Rails with the "Asset Pipeline" enabled, this helper will generate a link to /assets/stylesheets/. This link is then processed by the Sprockets gem. A stylesheet file can be stored in one of three locations: app/assets, lib/assets or vendor/assets.

You can specify a full path relative to the document root, or a URL. For example, to link to a stylesheet file that is inside a directory called stylesheets inside of one of app/assets, lib/assets or vendor/assets, you would do this:

```
<%= stylesheet_link_tag "main" %>
```

To include app/assets/stylesheets/main.css and app/assets/stylesheets/columns.css:

```
<%= stylesheet_link_tag "main", "columns" %>
```

To include app/assets/stylesheets/main.css and app/assets/stylesheets/photos/columns.css:

```
<%= stylesheet_link_tag "main", "/photos/columns" %>
```

To include http://example.com/main.css:

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

By default, the stylesheet_link_tag creates links with media="screen" rel="stylesheet" type="text/css". You can override any of these defaults by specifying an appropriate option (:media, :rel, or :type):

```
<%= stylesheet_link_tag "main_print", :media => "print" %>
```

If the asset pipeline is disabled, the all option links every CSS file in public/stylesheets:

```
<%= stylesheet_link_tag :all %>
```

You can supply the :recursive option to link files in subfolders of public/stylesheets as well:

```
<%= stylesheet_link_tag :all, :recursive => true %>
```

If you're loading multiple CSS files, you can create a better user experience by combining multiple files into a single download. To make this happen in production, specify :cache => true in your stylesheet_link_tag:

```
<%= stylesheet_link_tag "main", "columns", :cache => true %>
```

By default, the combined file will be delivered as stylesheets/all.css. You can specify a location for the cached asset file instead:

```
<%= stylesheet_link_tag "main", "columns",
  :cache => 'cache/main/display' %>
```

You can even use dynamic paths such as cache/#{current_sitemain/display.

Linking to Images with the image_tag

The image_tag helper builds an HTML<img /> tag to the specified file. By default, files are loaded from public/images.

Note that you must specify the extension of the image. Previous versions of Rails would allow you to just use the image name and would append .png if no extension was given but Rails 3.0 does not.

```
<%= image_tag "header.png" %>
```

You can supply a path to the image if you like:

```
<%= image_tag "icons/delete.gif" %>
```

You can supply a hash of additional HTML options:

```
<%= image_tag "icons/delete.gif", {:height => 45} %>
```

You can also supply an alternate image to show on mouseover:

```
<%= image_tag "home.gif", :onmouseover => "menu/home_highlight.gif" %>
```

You can supply alternate text for the image which will be used if the user has images turned off in their browser. If you do not specify an alt text explicitly, it defaults to the file name of the file, capitalized and with no extension. For example, these two image tags would return the same code:

```
<%= image_tag "home.gif" %>
<%= image_tag "home.gif", :alt => "Home" %>
```

You can also specify a special size tag,
in the format “{width}x{height}”:

```
<%= image_tag "home.gif", :size => "50x20" %>
```

In addition to the above special tags, you can supply a final hash of standard HTML options, such as :class, :id or :name:

```
<%= image_tag "home.gif", :alt => "Go Home",
                          :id => "HomeImage",
                          :class => 'nav_bar' %>
```

Linking to Videos with the video_tag

The video_tag helper builds an HTML 5 <video> tag to the specified file. By default, files are loaded from public/videos.

```
<%= video_tag "movie.ogg" %>
```

Produces

```
<video src="/videos/movie.ogg" />
```

Like an image_tag you can supply a path, either absolute, or relative to the public/videos directory. Additionally you can specify the :size => "#{width#{height}" option just like an image_tag. Video tags can also have any of the HTML options specified at the end (id, class et al).

The video tag also supports all of the <video>HTML options through the HTML options hash, including:

- :poster => 'image_name.png', provides an image to put in place of the video before it starts playing.
- :autoplay => true, starts playing the video on page load.
- :loop => true, loops the video once it gets to the end.
- :controls => true, provides browser supplied controls for the user to interact with the video.
- :autobuffer => true, the video will pre load the file for the user on page load.

You can also specify multiple videos to play by passing an array of videos to the video_tag:

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

This will produce:

```
<video><source src="trailer.ogg" /><source src="movie.ogg" /></video>
```

Linking to Audio Files with the audio_tag

The audio_tag helper builds an HTML 5 <audio> tag to the specified file. By default, files are loaded from public/audios.

```
<%= audio_tag "music.mp3" %>
```

You can supply a path to the audio file if you like:

```
<%= audio_tag "music/first_song.mp3" %>
```

You can also supply a hash of additional options, such as :id, :class etc.

Like the video_tag, the audio_tag has special options:

- :autoplay => true, starts playing the audio on page load
- :controls => true, provides browser supplied controls for the user to interact with the audio.
- :autobuffer => true, the audio will pre load the file for the user on page load.

6.3.2 Understanding yield

Within the context of a layout, yield identifies a section where content from the view should be inserted. The simplest way to use this is to have a single yield, into which the entire contents of the view currently being rendered is inserted:

```
<html>
  <head>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

You can also create a layout with multiple yielding regions:

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

The main body of the view will always render into the unnamed yield. To render content into a named yield, you use the content_for method.

6.3.3 Using the content_for Method

The content_for method allows you to insert content into a named yield block in your layout. For example, this view would work with the layout that you just saw:

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

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

The result of rendering this page into the supplied layout would be this HTML:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

The content_for method is very helpful when your layout contains distinct regions such as sidebars and footers that should get their own blocks of content inserted. It's also useful for inserting tags that load page-specific JavaScript or css files into the header of an otherwise generic layout.

6.3.4 Using Partials

Partial templates – usually just called "partials" – are another device for breaking the rendering process into more manageable chunks. With a partial, you can move the code for rendering a particular piece of a response to its own file.

Naming Partials

To render a partial as part of a view, you use the render method within the view:

```
<%= render "menu" %>
```

This will render a file named _menu.html.erb at that point within the view being rendered. Note the leading underscore character: partials are named with a leading underscore to distinguish them from regular views, even though they are referred to without the underscore. This holds true even when you're pulling in a partial from another folder:

```
<%= render "shared/menu" %>
```

That code will pull in the partial
from app/views/shared/_menu.html.erb.

Using Partials to Simplify Views

One way to use partials is to treat them as the equivalent of subroutines: as a way to move details out of a view so that you can grasp what's going on more easily. For example, you might have a view that looked like this:

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
...

<%= render "shared/footer" %>
```

Here, the _ad_banner.html.erb and _footer.html.erb partials could contain content that is shared among many pages in your application. You don't need to see the details of these sections when you're concentrating on a particular page.

For content that is shared among all pages in your application, you can use partials directly from layouts.

Partial Layouts

A partial can use its own layout file, just as a view can use a layout. For example, you might call a partial like this:

```
<%= render :partial => "link_area", :layout => "graybar" %>
```

This would look for a partial named _link_area.html.erb and render it using the layout _graybar.html.erb. Note that layouts for partials follow the same leading-underscore naming as regular partials, and are placed in the same folder with the partial that they belong to (not in the master layouts folder).

Also note that explicitly specifying :partial is required when passing additional options such as :layout.

Passing Local Variables

You can also pass local variables into partials, making them even more powerful and flexible. For example, you can use this technique to reduce duplication between new and edit pages, while still keeping a bit of distinct content:

- new.html.erb

```
<h1>New zone</h1>
<%= error_messages_for :zone %>
<%= render :partial => "form", :locals => { :zone => @zone } %>
```

- edit.html.erb

```
<h1>Editing zone</h1>
<%= error_messages_for :zone %>
<%= render :partial => "form", :locals => { :zone => @zone } %>
```

- _form.html.erb

```
<%= form_for(zone) do |f| %>
  <p>
    <b>Zone name</b><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Although the same partial will be rendered into both views, Action View's submit helper will return "Create Zone" for the new action and "Update Zone" for the edit action.

Every partial also has a local variable with the same name as the partial (minus the underscore). You can pass an object in to this local variable via the :object option:

```
<%= render :partial => "customer", :object => @new_customer %>
```

Within the customer partial, the customer variable will refer to @new_customer from the parent view.

In previous versions of Rails, the default local variable would look for an instance variable with the same name as the partial in the parent. This behavior was deprecated in 2.3 and has been removed in Rails 3.0.

If you have an instance of a model to render into a partial, you can use a shorthand syntax:

```
<%= render @customer %>
```

Assuming that the @customer instance variable contains an instance of the Customer model, this will use _customer.html.erb to render it and will pass the local variable customer into the partial which will refer to the @customer instance variable in the parent view.

Rendering Collections

Partials are very useful in rendering collections. When you pass a collection to a partial via the :collection option, the partial will be inserted once for each member in the collection:

- index.html.erb

```
<h1>Products</h1>
<%= render :partial => "product", :collection => @products %>
```

- _product.html.erb

```
<p>Product Name: <%= product.name %></p>
```

When a partial is called with a pluralized collection, then the individual instances of the partial have access to the member of the collection being rendered via a variable named after the partial. In this case, the partial is _product, and within the _product partial, you can refer to product to get the instance that is being rendered.

In Rails 3.0, there is also a shorthand for this. Assuming @products is a collection of product instances, you can simply write this in the index.html.erb to produce the same result:

```
<h1>Products</h1>
<%= render @products %>
```

Rails determines the name of the partial to use by looking at the model name in the collection. In fact, you can even create a heterogeneous collection and render it this way, and Rails will choose the proper partial for each member of the collection:
In the event that the collection is empty, render will return nil, so it should be fairly simple to provide alternative content.

```
<h1>Products</h1>
<%= render(@products) || 'There are no products available.' %>
```

- index.html.erb

```
<h1>Contacts</h1>
<%= render [customer1, employee1, customer2, employee2] %>
```

- customers/_customer.html.erb

```
<p>Customer: <%= customer.name %></p>
```

- employees/_employee.html.erb

```
<p>Employee: <%= employee.name %></p>
```

In this case, Rails will use the customer or employee partials as appropriate for each member of the collection.

Local Variables

To use a custom local variable name within the partial, specify the :as option in the call to the partial:

```
<%= render :partial => "product",
  :collection => @products, :as => :item %>
```

With this change, you can access an instance of the @products collection as the item local variable within the partial.

You can also pass in arbitrary local variables to any partial you are rendering with the :locals => {} option:

```
<%= render :partial => 'products', :collection => @products,
          :as => :item, :locals => {:title => "Products Page"} %>
```

Would render a partial _products.html.erb once for each instance of product in the @products instance variable passing the instance to the partial as a local variable called item and to each partial, make the local variable title available with the value Products Page.

Rails also makes a counter variable available within a partial called by the collection, named after the member of the collection followed by _counter. For example, if you're rendering @products, within the partial you can refer to product_counter to tell you how many times the partial has been rendered. This does not work in conjunction with the :as => :value option.

You can also specify a second partial to be rendered between instances of the main partial by using the :spacer_template option:

Spacer Templates

```
<%= render :partial => @products, :spacer_template => "product_ruler" %>
```

Rails will render the _product_ruler partial (with no data passed in to it) between each pair of _product partials.

6.3.5 Using Nested Layouts

You may find that your application requires a layout that differs slightly from your regular application layout to support one particular controller. Rather than repeating the main layout and editing it, you can accomplish this by using nested layouts (sometimes called sub-templates). Here's an example:

Suppose you have the following ApplicationController layout:

- app/views/layouts/application.html.erb

```
<html>
<head>
  <title><%= @page_title or 'Page Title' %></title>
  <%= stylesheet_link_tag 'layout' %>
  <style type="text/css"><%= yield :stylesheets %></style>
</head>
<body>
  <div id="top_menu">Top menu items here</div>
  <div id="menu">Menu items here</div>
  <div id="content">
    <%=content_for?(:content)?yield(:content):yield %>
  </div>
</body>
</html>
```

On pages generated by NewsController, you want to hide the top menu and add a right menu:

- app/views/layouts/news.html.erb

```
<% content_for :stylesheets do %>
  #top_menu {display: none}
  #right_menu {float: right; background-color: yellow; color: black}
<% end %>
<% content_for :content do %>
  <div id="right_menu">Right menu items here</div>
  <%= content_for?(:news_content) ? yield(:news_content) : yield %>
<% end %>
<%= render :template => 'layouts/application' %>
```

That's it. The News views will use the new layout, hiding the top menu and adding a new right menu inside the "content" div.

There are several ways of getting similar results with different sub-templating schemes using this technique. Note that there is no limit in nesting levels. One can use the ActionView::render method via render :template => 'layouts/news' to base a new layout on the News layout. If you are sure you will not subtemplate the News layout, you can replace the content_for?(:news_content) ? yield(:news_content) : yield with simply yield.

Chapter 7

Rails Form helpers

Forms in web applications are an essential interface for user input. However, form markup can quickly become tedious to write and maintain because of form control naming and their numerous attributes. Rails deals away with these complexities by providing view helpers for generating form markup. However, since they have different use-cases, developers are required to know all the differences between similar helper methods before putting them to use.

In this guide you will:

- Create search forms and similar kind of generic forms not representing any specific model in your application
- Make model-centric forms for creation and editing of specific database records
- Generate select boxes from multiple types of data
- Understand the date and time helpers Rails provides
- Learn what makes a file upload form different
- Learn some cases of building forms to external resources
- Find out where to look for complex forms

This guide is not intended to be a complete documentation of available form helpers and their arguments. Please visit the Rails API documentation for a complete reference.

7.1 Dealing with Basic Forms

The most basic form helper is form_tag.

```
<%= form_tag do %>
  Form contents
<% end %>
```

When called without arguments like this, it creates a <form> tag which, when submitted, will POST to the current page. For instance, assuming the current page is /home/index, the generated HTML will look like this (some line breaks added for readability):

```
<form accept-charset="UTF-8" action="/home/index" method="post">
  <div style="margin:0;padding:0">
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden"
           value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  Form contents
</form>
```

Now, you'll notice that the HTML contains something extra: a div element with two hidden input elements inside. This div is important, because the form cannot be successfully submitted without it. The first input element with name utf8 enforces browsers to properly respect your form's character encoding and is generated for all forms whether their actions are "GET" or "POST". The second input element with name authenticity_token is a security feature of Rails called **cross-site request forgery protection**, and form helpers generate it for every non-GET form (provided that this security feature is enabled). You can read more about this in the Security Guide.

Throughout this guide, the div with the hidden input elements will be excluded from code samples for brevity.

7.1.1 A Generic Search Form

One of the most basic forms you see on the web is a search form. This form contains:

1. a form element with "GET" method,
2. a label for the input,
3. a text input element, and
4. a submit element.

To create this form you will use form_tag, label_tag, text_field_tag, and submit_tag, respectively. Like this:

```
<%= form_tag("/search", :method => "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/search" method="get">
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

For every form input, an ID attribute is generated from its name ("q" in the example). These IDs can be very useful for CSS styling or manipulation of form controls with JavaScript.

Besides text_field_tag and submit_tag, there is a similar helper for every form control in HTML.

Always use "GET" as the method for search forms. This allows users to bookmark a specific search and get back to it. More generally Rails encourages you to use the right HTTP verb for an action.

7.1.2 Multiple Hashes in Form Helper Calls

The form_tag helper accepts 2 arguments: the path for the action and an options hash. This hash specifies the method of form submission and HTML options such as the form element's class.

As with the link_to helper, the path argument doesn't have to be given a string; it can be a hash of URL parameters recognizable by Rails' routing mechanism, which will turn the hash into a valid URL. However, since both arguments to form_tag are hashes, you can easily run into a problem if you would like to specify both. For instance, let's say you write this:

```
form_tag(:controller => "people", :action => "search",
        :method => "get", :class => "nifty_form")
# => '<form accept-charset="UTF-8"
            action="/people/search?method=get&class=nifty_form"
            method="post">'
```

Here, method and class are appended to the query string of the generated URL because you even though you mean to write two hashes, you really only specified one. So you need to tell Ruby which is which by delimiting the first hash (or both) with curly brackets. This will generate the HTML you expect:

```
form_tag({:controller => "people", :action => "search"},
         :method => "get", :class => "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search"
            method="get" class="nifty_form">'
```

7.1.3 Helpers for Generating Form Elements

Rails provides a series of helpers for generating form elements such as checkboxes, text fields, and radio buttons. These basic helpers, with names ending in "_tag" (such as text_field_tag and check_box_tag), generate just a single <input> element. The first parameter to these is always the name of the input. When the form is submitted, the name will be passed along with the form data, and will make its way to the params hash in the controller with the value entered by the user for that field. For example, if the form contains <%= text_field_tag(:query) %>, then you would be able to get the value of this field in the controller with params[:query].

When naming inputs, Rails uses certain conventions that make it possible to submit parameters with non-scalar values such as arrays or hashes, which will also be accessible in params. You can read more about them in chapter 7 of this guide. For details on the precise usage of these helpers, please refer to the API documentation.

Checkboxes

Checkboxes are form controls that give the user a set of options they can enable or disable:

```
<%= check_box_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= check_box_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

This generates the following:

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
<label for="pet_cat">I own a cat</label>
```

The first parameter to check_box_tag, of course, is the name of the input. The second parameter, naturally, is the value of the input. This value will be included in the form data (and be present in params) when the checkbox is checked.

Radio Buttons

Radio buttons, while similar to checkboxes, are controls that specify a set of options in which they are mutually exclusive (i.e., the user can only pick one):

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 21") %>
```

Output:

```
<input id="age_child" name="age" type="radio" value="child" />
<label for="age_child">I am younger than 21</label>
<input id="age_adult" name="age" type="radio" value="adult" />
<label for="age_adult">I'm over 21</label>
```

As with check_box_tag, the second parameter to radio_button_tag is the value of the input. Because these two radio buttons share the same name (age) the user will only be able to select one, and params[:age] will contain either “child” or “adult”.

Always use labels for checkbox and radio buttons. They associate text with a specific option and make it easier for users to click the inputs by expanding the clickable region.

7.1.4 Other Helpers of Interest

Other form controls worth mentioning are textareas, password fields, hidden fields, search fields, telephone fields, URL fields and email fields:

```
<%= text_area_tag(:message, "Hi, nice site", :size => "24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
```

Output:

```
<textarea id="message" name="message" cols="24" rows="6">
  Hi, nice site
```

```
</textarea>
<input id="password" name="password" type="password" />
<input id="parent_id" name="parent_id" type="hidden" value="5" />
<input id="user_name" name="user[name]" size="30" type="search" />
<input id="user_phone" name="user[phone]" size="30" type="tel" />
<input id="user_homepage" size="30" name="user[homepage]" type="url" />
<input id="user_address" size="30" name="user[address]" type="email" />
```

Hidden inputs are not shown to the user but instead hold data like any textual input. Values inside them can be changed with JavaScript.

The search, telephone, URL, and email inputs are HTML5 controls. If you require your app to have a consistent experience in older browsers, you will need an HTML5 polyfill (provided by CSS and/or JavaScript). There is definitely no shortage of solutions for this, although a couple of popular tools at the moment are Modernizr and yepnope, which provide a simple way to add functionality based on the presence of detected HTML5 features.

If you're using password input fields (for any purpose), you might want to configure your application to prevent those parameters from being logged. You can learn about this in the Security Guide.

7.2 Dealing with Model Objects

7.2.1 Model Object Helpers

A particularly common task for a form is editing or creating a model object. While the *_tag helpers can certainly be used for this task they are somewhat verbose as for each tag you would have to ensure the correct parameter name is used and set the default value of the input appropriately. Rails provides helpers tailored to this task. These helpers lack the _tag suffix, for example text_field, text_area.

For these helpers the first argument is the name of an instance variable and the second is the name of a method (usually an attribute) to call on that object. Rails will set the value of the input control to the return value of that method for the object and set an appropriate input name. If your controller has defined @person and that person's name is Henry then a form containing:

```
<%= text_field(:person, :name) %>
```

will produce output similar to

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

Upon form submission the value entered by the user will be stored in params[:person][:name]. The params[:person] hash is suitable for passing to Person.new or, if @person is an instance of Person, @person.update_attributes. While the name of an attribute is the most common second parameter to these helpers this is not compulsory. In the example above, as long as person objects have a name and a name= method Rails will be happy.

You must pass the name of an instance variable, i.e. :person or "person", not an actual instance of your model object.

Rails provides helpers for displaying the validation errors associated with a model object. These are covered in detail by the Active Record Validations and Callbacks guide.

7.2.2 Binding a Form to an Object

While this is an increase in comfort it is far from perfect. If Person has many attributes to edit then we would be repeating the name of the edited object many times. What we want to do is somehow bind a form to a model object, which is exactly what form_for does.

Assume we have a controller for dealing with articles app/controllers/articles_controller.rb:

```
def new
  @article = Article.new
end
```

The corresponding view app/views/articles/new.html.erb using form_for looks like this:

```
<%= form_for @article, :url => { :action => "create" },
    :html => {:class => "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, :size => "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```

There are a few things to note here:

1. @article is the actual object being edited.
2. There is a single hash of options. Routing options are passed in the :url hash, HTML options are passed in the :html hash. Also you can provide a :namespace option for your form to ensure uniqueness of id attributes on form elements. The namespace attribute will be prefixed with underscore on the generated HTML id.
3. The form_for method yields a **form builder** object (the f variable).
4. Methods to create form controls are called **on** the form builder object f

The resulting HTML is:

```
<form accept-charset="UTF-8" action="/articles/create"
                      method="post" class="nifty_form">
<input id="article_title" name="article[title]" size="30" type="text" />
<textarea id="article_body" name="article[body]" cols="60" rows="12">
</textarea>
<input name="commit" type="submit" value="Create" />
</form>
```

The name passed to form_for controls the key used in params to access the form's values. Here the name is article and so all the inputs have names of the form article[attribute_name]. Accordingly, in the create action params[:article] will be a hash with keys :title and :body. You can read more about the significance of input names in the parameter_names section.

The helper methods called on the form builder are identical to the model object helpers except that it is not necessary to specify which object is being edited since this is already managed by the form builder.

You can create a similar binding without actually creating <form> tags with the fields_for helper. This is useful for editing additional model objects with the same form. For example if you had a Person model with an associated ContactDetail model you could create a form for creating both like so:

```
<%= form_for @person, :url => { :action => "create" } do |person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for @person.contact_detail do |contact_details_form| %>
    <%= contact_details_form.text_field :phone_number %>
  <% end %>
<% end %>
```

which produces the following output:

```
<form accept-charset="UTF-8" action="/people/create"
    class="new_person" id="new_person" method="post">
  <input id="person_name" name="person[name]" size="30" type="text" />
  <input id="contact_detail_phone_number"
      name="contact_detail[phone_number]" size="30" type="text" />
</form>
```

The object yielded by fields_for is a form builder like the one yielded by form_for (in fact form_for calls fields_for internally).

7.2.3 Relying on Record Identification

The Article model is directly available to users of the application, so — following the best practices for developing with Rails — you should declare it **a resource**:

```
resources :articles
```

Declaring a resource has a number of side-affects. See Rails Routing From the Outside In for more information on setting up and using resources.

When dealing with RESTful resources, calls to form_for can get significantly easier if you rely on **record identification**. In short, you can just pass the model instance and have Rails figure out model name and the rest:

```
## Creating a new article
# long-style:
form_for(@article, :url => articles_path)
# same thing, short-style (record identification gets used):
form_for(@article)

## Editing an existing article
# long-style:
form_for(@article, :url => article_path(@article),
                   :html => { :method => "put" })
# short-style:
form_for(@article)
```

Notice how the short-style form_for invocation is conveniently the same, regardless of the record being new or existing. Record identification is smart enough to figure out if the record is new by asking record.new_record?. It also selects the correct path to submit to and the name based on the class of the object.

Rails will also automatically set the class and id of the form appropriately: a form creating an article would have id and class-new_article. If you were editing the article with id 23, the class would be set to edit_article and the id to edit_article_23. These attributes will be omitted for brevity in the rest of this guide.

When you're using STI (single-table inheritance) with your models, you can't rely on record identification on a subclass if only their parent class is declared a resource. You will have to specify the model name, :url, and :method explicitly.

Dealing with Namespaces

If you have created namespaced routes, form_for has a nifty shorthand for that too. If your application has an admin namespace then

```
form_for [:admin, @article]
```

will create a form that submits to the articles controller inside the admin namespace (submitting to admin_article_path(@article) in the case of an update). If you have several levels of namespacing then the syntax is similar:

```
form_for [:admin, :management, @article]
```

For more information on Rails' routing system and the associated conventions, please see the routing guide.

7.2.4 How do forms with PUT or DELETE methods work?

The Rails framework encourages RESTful design of your applications, which means you'll be making a lot of "PUT" and "DELETE" requests (besides "GET" and "POST"). However, most browsers don't support methods other than "GET" and "POST" when it comes to submitting forms.

Rails works around this issue by emulating other methods over POST with a hidden input named "_method", which is set to reflect the desired method:

```
form_tag(search_path, :method => "put")
```

output:

```
<form accept-charset="UTF-8" action="/search" method="post">
  <div style="margin:0;padding:0">
    <input name="_method" type="hidden" value="put" />
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden"
           value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  ...
```

When parsing POSTed data, Rails will take into account the special _method parameter and acts as if the HTTP method was the one specified inside it ("PUT" in this example).

7.3 Making Select Boxes with Ease

Select boxes in HTML require a significant amount of markup (one OPTION element for each option to choose from), therefore it makes the most sense for them to be dynamically generated.

Here is what the markup might look like:

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

Here you have a list of cities whose names are presented to the user. Internally the application only wants to handle their IDs so they are used as the options' value attribute. Let's see how Rails can help out here.

7.3.1 The Select and Option Tags

The most generic helper is select_tag, which — as the name implies — simply generates the SELECT tag that encapsulates an options string:

```
<%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

This is a start, but it doesn't dynamically create the option tags. You can generate option tags with the options_for_select helper:

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...]) %>
```

```
output:

<option value="1">Lisbon</option>
<option value="2">Madrid</option>
...
```

The first argument to options_for_select is a nested array where each element has two elements: option text (city name) and option value (city id). The option value is what will be submitted to your controller. Often this will be the id of a corresponding database object but this does not have to be the case.

Knowing this, you can combine select_tag and options_for_select to achieve the desired, complete markup:

```
<%= select_tag(:city_id, options_for_select(...)) %>
```

options_for_select allows you to pre-select an option by passing its value.

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...], 2) %>

output:

<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...
```

Whenever Rails sees that the internal value of an option being generated matches this value, it will add the selected attribute to that option.

The second argument to options_for_select must be exactly equal to the desired internal value. In particular if the value is the integer 2 you cannot pass "2" to options_for_select — you must pass 2. Be aware of values extracted from the params hash as they are all strings.

7.3.2 Select Boxes for Dealing with Models

In most cases form controls will be tied to a specific database model and as you might expect Rails provides helpers tailored for that purpose. Consistent with other form helpers, when dealing with models you drop the _tag suffix from select_tag:

```
# controller:
@person = Person.new(:city_id => 2)


# view:
<%= select(:person, :city_id, [['Lisbon', 1], ['Madrid', 2], ...]) %>
```

Notice that the third parameter, the options array, is the same kind of argument you pass to options_for_select. One advantage here is that you don't have to worry about pre-selecting the correct city if the user already has one — Rails will do this for you by reading from the @person.city_id attribute.

As with other helpers, if you were to use the select helper on a form builder scoped to the @person object, the syntax would be:

```
# select on a form builder
<%= f.select(:city_id, ...) %>
```

If you are using select (or similar helpers such as collection_select, select_tag) to set a belongs_to association you must pass the name of the foreign key (in the example above city_id), not the name of association itself. If you specify city instead of city_id Active Record will raise an error along the lines of ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750) when you pass the params hash to Person.new or update_attributes. Another way of looking at this is that form helpers only edit attributes. You should also be aware of the potential security ramifications of allowing users to edit foreign keys directly. You may wish to consider the use of attr_protected and attr_accessible. For further details on this, see the Ruby On Rails Security Guide.

7.3.3 Option Tags from a Collection of Arbitrary Objects

Generating options tags with options_for_select requires that you create an array containing the text and value for each option. But what if you had a City model (perhaps an Active Record one) and you wanted to generate option tags from a collection of those objects? One solution would be to make a nested array by iterating over them:

```
<% cities_array = City.all.map { |city| [city.name, city.id] } %>
<%= options_for_select(cities_array) %>
```

This is a perfectly valid solution, but Rails provides a less verbose alternative: options_from_collection_for_select. This helper expects a collection of arbitrary objects and two additional arguments: the names of the methods to read the option **value** and **text** from, respectively:

```
<%= options_from_collection_for_select(City.all, :id, :name) %>
```

As the name implies, this only generates option tags. To generate a working select box you would need to use it in conjunction with select_tag, just as you would with options_for_select. When working with model objects, just as select combines select_tag and options_for_select, collection_select combines select_tag with options_from_collection_for_select.

```
<%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

To recap, options_from_collection_for_select is to collection_select what options_for_select is to select.

Pairs passed to options_for_select should have the name first and the id second, however with options_from_collection_for_select the first argument is the value method and the second the text method.

7.3.4 Time Zone and Country Select

To leverage time zone support in Rails, you have to ask your users what time zone they are in. Doing so would require generating select options from a list of pre-defined TimeZone objects using collection_select, but you can simply use the time_zone_select helper that already wraps this:

```
<%= time_zone_select(:person, :time_zone) %>
```

There is also time_zone_options_for_select helper for a more manual (therefore more customizable) way of doing this. Read the API documentation to learn about the possible arguments for these two methods.

Rails used to have a country_select helper for choosing countries, but this has been extracted to the country_select plugin. When using this, be aware that the exclusion or inclusion of certain names from the list can be somewhat controversial (and was the reason this functionality was extracted from Rails).

7.4 Using Date and Time Form Helpers

The date and time helpers differ from all the other form helpers in two important respects:

1. Dates and times are not representable by a single input element. Instead you have several, one for each component (year, month, day etc.) and so there is no single value in your params hash with your date or time.

2. Other helpers use the _tag suffix to indicate whether a helper is a barebones helper or one that operates on model objects. With dates and times, select_date, select_time and select_datetime are the barebones helpers, date_select, time_select and datetime_select are the equivalent model object helpers.

Both of these families of helpers will create a series of select boxes for the different components (year, month, day etc.).

7.4.1 Barebones Helpers

The select_* family of helpers take as their first argument an instance of Date, Time or DateTime that is used as the currently selected value. You may omit this parameter, in which case the current date is used. For example

```
<%= select_date Date.today, :prefix => :start_date %>
```

outputs (with actual option values omitted for brevity)

```
<select id="start_date_year" name="start_date[year]"> ... </select>
<select id="start_date_month" name="start_date[month]"> ... </select>
<select id="start_date_day" name="start_date[day]"> ... </select>
```

The above inputs would result in params[:start_date] being a hash with keys :year, :month, :day. To get an actual Time or Date object you would have to extract these values and pass them to the appropriate constructor, for example

```
Date.civil(params[:start_date][:year].to_i,
           params[:start_date][:month].to_i,
           params[:start_date][:day].to_i)
```

The :prefix option is the key used to retrieve the hash of date components from the params hash. Here it was set to start_date, if omitted it will default to date.

7.4.2 Model Object Helpers

select_date does not work well with forms that update or create Active Record objects as Active Record expects each element of the params hash to correspond to one attribute. The model object helpers for dates and times submit parameters with special names, when Active Record sees parameters with such names it knows they must be combined with the other parameters and given to a constructor appropriate to the column type. For example:

```
<%= date_select :person, :birth_date %>
```

outputs (with actual option values omitted for brevity)

```
<select id="person_birth_date_1i" name="person[birth_date(1i)]">
</select>
<select id="person_birth_date_2i" name="person[birth_date(2i)]">
</select>
<select id="person_birth_date_3i" name="person[birth_date(3i)]">
</select>
```

which results in a params hash like

```
{:person => {'birth_date(1i)' => '2008',
'birth_date(2i)' => '11', 'birth_date(3i)' => '22'}}
```

When this is passed to Person.new (or update_attributes), Active Record spots that these parameters should all be used to construct the birth_date attribute and uses the suffixed information to determine in which order it should pass these parameters to functions such as Date.civil.

7.4.3 Common Options

Both families of helpers use the same core set of functions to generate the individual select tags and so both accept largely the same options. In particular, by default Rails will generate year options 5 years either side of the current year. If this is not an appropriate range, the :start_year and :end_year options override this. For an exhaustive list of the available options, refer to the API documentation.

As a rule of thumb you should be using date_select when working with model objects and select_date in other cases, such as a search form which filters results by date.

In many cases the built-in date pickers are clumsy as they do not aid the user in working out the relationship between the date and the day of the week.

7.4.4 Individual Components

Occasionally you need to display just a single date component such as a year or a month. Rails provides a series of helpers for this, one for each component select_year, select_month, select_day, select_hour, select_minute, select_second. These helpers are fairly straightforward. By default they will generate an input field named after the time component (for example "year" for select_year, "month" for select_month etc.) although this can be overridden with the :field_name option. The :prefix option works in the same way that it does for select_date and select_time and has the same default value.

The first parameter specifies which value should be selected and can either be an instance of a Date, Time or DateTime, in which case the relevant component will be extracted, or a numerical value. For example

```
<%= select_year(2009) %>
<%= select_year(Time.now) %>
```

will produce the same output if the current year is 2009 and the value chosen by the user can be retrieved by params[:date][:year].

7.5 Uploading Files

A common task is uploading some sort of file, whether it's a picture of a person or a CSV file containing data to process. The most important thing to remember with file uploads is that the rendered form's encoding **MUST** be set to "multipart/form-data". If you use form_for, this is done automatically. If you use form_tag, you must set it yourself, as per the following example.

The following two forms both upload a file.

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

```
<%= form_for @person do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Since Rails 3.1, forms rendered using form_for have their encoding set to multipart/form-data automatically once a file_field is used inside the block. Previous versions required you to set this explicitly.

Rails provides the usual pair of helpers: the barebones file_field_tag and the model oriented file_field. The only difference with other helpers is that you cannot set a default value for file inputs as this would have no meaning. As you would expect in the first case the uploaded file is in params[:picture] and in the second case in params[:person][:picture].

7.5.1 What Gets Uploaded

The object in the params hash is an instance of a subclass of IO. Depending on the size of the uploaded file it may in fact be a StringIO or an instance of File backed by a temporary file. In both cases the object will have an original_filename attribute containing the name the file had on the user's computer and a content_type attribute containing the MIME type of the uploaded file. The following snippet saves the uploaded content in #{Rails.rootpublic/uploads under the same name as the original file (assuming the form was the one in the previous example).

```
def upload
  uploaded_io = params[:person][:picture]
  File.open(Rails.root.join('public', 'uploads',
       uploaded_io.original_filename), 'w') do |file|
    file.write(uploaded_io.read)
  end
end
```

Once a file has been uploaded, there are a multitude of potential tasks, ranging from where to store the files (on disk, Amazon S3, etc) and associating them with models to resizing image files and generating thumbnails. The intricacies of this are beyond the scope of this guide, but there are several libraries designed to assist with these. Two of the better known ones are CarrierWave and Paperclip.

If the user has not selected a file the corresponding parameter will be an empty string.

7.5.2 Dealing with Ajax

Unlike other forms making an asynchronous file upload form is not as simple as providing form_for with :remote => true. With an Ajax form the serialization is done by JavaScript running inside the browser and since JavaScript cannot read files from your hard drive the file cannot be uploaded. The most common workaround is to use an invisible iframe that serves as the target for the form submission.

7.6 Customizing Form Builders

As mentioned previously the object yielded by form_for and fields_for is an instance of FormBuilder (or a subclass thereof). Form builders encapsulate the notion of displaying form elements for a single object. While you can of course write helpers for your forms in the usual way you can also subclass FormBuilder and add the helpers there. For example

```
<%= form_for @person do |f| %>
  <%= text_field_with_label f, :first_name %>
<% end %>
```

can be replaced with

```
<%= form_for @person, :builder => LabellingFormBuilder do |f| %>
  <%= f.text_field :first_name %>
<% end %>
```

by defining a LabellingFormBuilder class similar to the following:

```
class LabellingFormBuilder < ActionView::Helpers::FormBuilder
  def text_field(attribute, options={})
    label(attribute) + super
  end
end
```

If you reuse this frequently you could define a labeled_form_for helper that automatically applies the :builder => LabellingFormBuilder option.

The form builder used also determines what happens when you do

```
<%= render :partial => f %>
```

If f is an instance of FormBuilder then this will render the form partial, setting the partial's object to the form builder. If the form builder is of class LabellingFormBuilder then the labelling_form partial would be rendered instead.

7.7 Understanding Parameter Naming Conventions

As you've seen in the previous sections, values from forms can be at the top level of the params hash or nested in another hash. For example in a standard create action for a Person model, params[:model] would usually be a hash of all the attributes for the person to create. The params hash can also contain arrays, arrays of hashes and so on.

Fundamentally HTML forms don't know about any sort of structured data, all they generate is name–value pairs, where pairs are just plain strings. The arrays and hashes you see in your application are the result of some parameter naming conventions that Rails uses.

You may find you can try out examples in this section faster by using the console to directly invoke Racks' parameter parser. For example,

```
Rack::Utils.parse_query "name=fred&phone=0123456789"
# => {"name"=>"fred", "phone"=>"0123456789"}
```

7.7.1 Basic Structures

The two basic structures are arrays and hashes. Hashes mirror the syntax used for accessing the value in params. For example if a form contains

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

the params hash will contain

```
{'person' => {'name' => 'Henry'}}
```

and params[:person][:name] will retrieve the submitted value in the controller.

Hashes can be nested as many levels as required, for example

```
<input id="person_address_city" name="person[address][city]"
                                type="text" value="New York"/>
```

will result in the params hash being

```
{'person' => {'address' => {'city' => 'New York'}}}
```

Normally Rails ignores duplicate parameter names. If the parameter name contains an empty set of square brackets [] then they will be accumulated in an array. If you wanted people to be able to input multiple phone numbers, you could place this in the form:

```
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
```

This would result in params[:person][:phone_number] being an array.

7.7.2 Combining Them

We can mix and match these two concepts. For example, one element of a hash might be an array as in the previous example, or you can have an array of hashes. For example a form might let you create any number of addresses by repeating the following form fragment

```
<input name="addresses[][line1]" type="text"/>
<input name="addresses[][line2]" type="text"/>
<input name="addresses[][city]" type="text"/>
```

This would result in params[:addresses] being an array of hashes with keys line1, line2 and city. Rails decides to start accumulating values in a new hash whenever it encounters an input name that already exists in the current hash.

There's a restriction, however, while hashes can be nested arbitrarily, only one level of "arrayness" is allowed. Arrays can be usually replaced by hashes, for example instead of having an array of model objects one can have a hash of model objects keyed by their id, an array index or some other parameter.

Array parameters do not play well with the check_box helper. According to the HTML specification unchecked checkboxes submit no value. However it is often convenient for a checkbox to always submit a value. The check_box helper fakes this by creating an auxiliary hidden input with the same name. If the checkbox is unchecked only the hidden input is submitted and if it is checked then both are submitted but the value submitted by the checkbox takes precedence. When working with array parameters this duplicate submission will confuse Rails since duplicate input names are how it decides when to start a new array element. It is preferable to either use check_box_tag or to use hashes instead of arrays.

7.7.3 Using Form Helpers

The previous sections did not use the Rails form helpers at all. While you can craft the input names yourself and pass them directly to helpers such as text_field_tag Rails also provides higher level support. The two tools at your disposal here are the name parameter to form_for and fields_for and the :index option that helpers take.

You might want to render a form with a set of edit fields for each of a person's addresses. For example:

```
<%= form_for @person do |person_form| %>
  <%= person_form.text_field :name %>
  <% @person.addresses.each do |address| %>
  <%= person_form.fields_for address, :index => address do |address_form|%>
      <%= address_form.text_field :city %>
  <% end %>
  <% end %>
<% end %>
```

Assuming the person had two addresses, with ids 23 and 45 this would create output similar to this:

```
<form accept-charset="UTF-8" action="/people/1"
      class="edit_person" id="edit_person_1" method="post">
<input id="person_name" name="person[name]" size="30" type="text" />
<input id="person_address_23_city"
  name="person[address][23][city]" size="30" type="text" />
<input id="person_address_45_city"
  name="person[address][45][city]" size="30" type="text" />
</form>
```

This will result in a params hash that looks like

```
{'person' => {'name' => 'Bob', 'address' =>
    {'23' => {'city' => 'Paris'}, '45' => {'city' => 'London'}}}}
```

Rails knows that all these inputs should be part of the person hash because you called fields_for on the first form builder. By specifying an :index option you're telling Rails that instead of naming the inputs person[address][city] it should insert that index surrounded by [] between the address and the city. If you pass an Active Record object as we did then Rails will call to_param on it, which by default returns the database id. This is often useful as it is then easy to locate which Address record should be modified. You can pass numbers with some other significance, strings or even nil (which will result in an array parameter being created).

To create more intricate nestings, you can specify the first part of the input name (person[address] in the previous example) explicitly, for example

```
<%= fields_for 'person[address][primary]', address,
              :index => address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

will create inputs like

```
<input id="person_address_primary_1_city"
       name="person[address][primary][1][city]"
       size="30" type="text" value="bologna" />
```

As a general rule the final input name is the concatenation of the name given to fields_for/form_for, the index value and the name of the attribute. You can also pass an :index option directly to helpers such as text_field, but it is usually less repetitive to specify this at the form builder level rather than on individual input controls.

As a shortcut you can append [] to the name and omit the :index option. This is the same as specifying :index => address so

```
<%= fields_for 'person[address][primary][]', address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

produces exactly the same output as the previous example.

7.8 Forms to external resources

If you need to post some data to an external resource it is still great to build your from using rails form helpers. But sometimes you need to set an authenticity_token for this resource. You can do it by passing an :authenticity_token => 'your_external_token' parameter to the form_tag options:

```
<%= form_tag 'http://farfar.away/form',
    :authenticity_token => 'external_token') do %>
  Form contents
<% end %>
```

Sometimes when you submit data to an external resource, like payment gateway, fields you can use in your form are limited by an external API. So you may want not to generate an authenticity_token hidden field at all. For doing this just pass false to the :authenticity_token option:

```
<%= form_tag 'http://farfar.away/form',
    :authenticity_token => false) do %>
  Form contents
<% end %>
```

The same technique is available for the form_for too:

```
<%= form_for @invoice, :url => external_url,
    :authenticity_token => 'external_token' do |f|
  Form contents
<% end %>
```

Or if you don't want to render an authenticity_token field:

```
<%= form_for @invoice, :url => external_url,
    :authenticity_token => false do |f|
  Form contents
<% end %>
```

7.9 Building Complex Forms

Many apps grow beyond simple forms editing a single object. For example when creating a Person you might want to allow the user to (on the same form) create multiple address records (home, work, etc.). When later editing that person the user should be able to add, remove or amend addresses as necessary. While this guide has shown you all the pieces necessary to handle this, Rails does not yet have a standard end-to-end way of accomplishing this, but many have come up with viable approaches. These include:

- As of Rails 2.3, Rails includes Nested Attributes and Nested Object Forms
- Ryan Bates' series of Railscasts on complex forms
- Handle Multiple Models in One Form from Advanced Rails Recipes
- Eloy Duran's complex-forms-examples application
- Lance Ivy's nested_assignment plugin and sample application
- James Golick's attribute_fu plugin

Chapter 8

Action Controller Overview

In this guide you will learn how controllers work and how they fit into the request cycle in your application. After reading this guide, you will be able to:

- Follow the flow of a request through a controller
- Understand why and how to store data in the session or cookies
- Work with filters to execute code during request processing
- Use Action Controller's built-in HTTP authentication
- Stream data directly to the user's browser
- Filter sensitive parameters so they do not appear in the application's log
- Deal with exceptions that may be raised during request processing

8.1 What Does a Controller Do?

Action Controller is the C in MVC. After routing has determined which controller to use for a request, your controller is responsible for making sense of the request and producing the appropriate output. Luckily, Action Controller does most of the groundwork for you and uses smart conventions to make this as straightforward as possible.

For most conventional RESTful applications, the controller will receive the request (this is invisible to you as the developer), fetch or save data from a model and use a view to create HTML output. If your controller needs to do things a little differently, that's not a problem, this is just the most common way for a controller to work.

A controller can thus be thought of as a middle man between models and views. It makes the model data available to the view so it can display that data to the user, and it saves or updates data from the user to the model.

For more details on the routing process, see Rails Routing from the Outside In.

8.2 Methods and Actions

A controller is a Ruby class which inherits from ApplicationController and has methods just like any other class. When your application receives a request, the routing will determine which controller and action to run, then Rails creates an instance of that controller and runs the method with the same name as the action.

```
class ClientsController < ApplicationController
  def new
  end
end
```

As an example, if a user goes to /clients/new in your application to add a new client, Rails will create an instance of ClientsController and run the new method. Note that the empty method from the example above could work just fine because Rails will by default render the new.html.erb view unless the action says otherwise. The new method could make available to the view a @client instance variable by creating a new Client:

```
def new
  @client = Client.new
end
```

The Layouts & Rendering Guide explains this in more detail.

ApplicationController inherits from ActionController::Base, which defines a number of helpful methods. This guide will cover some of these, but if you're curious to see what's in there, you can see all of them in the API documentation or in the source itself.

Only public methods are callable as actions. It is a best practice to lower the visibility of methods which are not intended to be actions, like auxiliary methods or filters.

8.3 Parameters

You will probably want to access data sent in by the user or other parameters in your controller actions. There are two kinds of parameters possible in a web application. The first are parameters that are sent as part of the URL, called query string parameters. The query string is everything after "?" in the URL. The second type of parameter is usually referred to as POST data. This information usually comes from an HTML form which has been filled in by the user. It's called POST data because it can only be sent as part of an HTTPPOST request. Rails does not make any distinction between query string parameters and POST parameters, and both are available in the params hash in your controller:

```
class ClientsController < ActionController::Base
  # This action uses query string parameters because it gets run
  # by an HTTP GET request, but this does not make any difference
  # to the way in which the parameters are accessed. The URL for
  # this action would look like this in order to list activated
  # clients: /clients?status=activated
  def index
    if params[:status] == "activated"
      @clients = Client.activated
    else
      @clients = Client.unactivated
    end
  end

  # This action uses POST parameters. They are most likely coming
  # from an HTML form which the user has submitted. The URL for
  # this RESTful request will be "/clients", and the data will be
  # sent as part of the request body.
  def create
    @client = Client.new(params[:client])
    if @client.save
      redirect_to @client
    else
      # This line overrides the default rendering behavior, which
      # would have been to render the "create" view.
      render :action => "new"
    end
  end
end
```

8.3.1 Hash and Array Parameters

The params hash is not limited to one-dimensional keys and values. It can contain arrays and (nested) hashes. To send an array of values, append an empty pair of square brackets "[]" to the key name:

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

The actual URL in this example will be encoded as "/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3" as "[" and "]" are not allowed in URLs. Most of the time you don't have to worry about this because the browser will take care of it for you, and Rails will decode it back when it receives it, but if you ever find yourself having to send those requests to the server manually you have to keep this in mind.

The value of params[:ids] will now be ["1", "2", "3"]. Note that parameter values are always strings; Rails makes no attempt to guess or cast the type.

To send a hash you include the key name inside the brackets:

```
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]" value="12345" />
  <input type="text" name="client[address][postcode]" value="12345" />
  <input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

When this form is submitted, the value of params[:client] will be {"name" => "Acme", "phone" => "12345", "address" => {"postcode" => "12345", "city" => "Carrot City"
. Note the nested hash in params[:client][:address].

Note that the params hash is actually an instance of HashWithIndifferentAccess from Active Support, which acts like a hash that lets you use symbols and strings interchangeably as keys.

8.3.2 JSON/XML parameters

If you're writing a web service application, you might find yourself more comfortable on accepting parameters in JSON or XML format. Rails will automatically convert your parameters into params hash, which you'll be able to access like you would normally do with form data.

So for example, if you are sending this JSON parameter:

```
{"company": { "name": "acme", "address": "123 Carrot Street" } }
```

You'll get params[:company] as {:name => "acme", "address" => "123 Carrot Street"}.

Also, if you've turned on config.wrap_parameters in your initializer or calling wrap_parameters in your controller, you can safely omit the root element in the JSON/XML parameter. The parameters will be cloned and wrapped in the key according to your controller's name by default. So the above parameter can be written as:

```
{ "name": "acme", "address": "123 Carrot Street" }
```

And assume that you're sending the data to CompaniesController, it would then be wrapped in :company key like this:

```
{ :name => "acme", :address => "123 Carrot Street",
:company => { :name => "acme", :address => "123 Carrot Street" }}
```

You can customize the name of the key or specific parameters you want to wrap by consulting the API documentation

8.3.3 Routing Parameters

The params hash will always contain the :controller and :action keys, but you should use the methods controller_name and action_name instead to access these values. Any other parameters defined by the routing, such as :id will also be available. As an example, consider a listing of clients where the list can show either active or inactive clients. We can add a route which captures the :status parameter in a "pretty" URL:

```
match '/clients/:status' => 'clients#index', :foo => "bar"
```

In this case, when a user opens the URL/clients/active, params[:status] will be set to "active". When this route is used, params[:foo] will also be set to "bar" just like it was passed in the query string. In the same way params[:action] will contain "index".

8.3.4 default_url_options

You can set global default parameters for URL generation by defining a method called default_url_options in your controller. Such a method must return a hash with the desired defaults, whose keys must be symbols:

```
class ApplicationController < ActionController::Base
  def default_url_options
    {:locale => I18n.locale}
  end
end
```

These options will be used as a starting point when generating URLs, so it's possible they'll be overridden by the options passed in url_for calls.

If you define default_url_options in ApplicationController, as in the example above, it would be used for all URL generation. The method can also be defined in one specific controller, in which case it only affects URLs generated there.

8.4 Session

Your application has a session for each user in which you can store small amounts of data that will be persisted between requests. The session is only available in the controller and the view and can use one of a number of different storage mechanisms:

- ActionDispatch::Session::CookieStore – Stores everything on the client.
- ActiveRecord::SessionStore – Stores the data in a database using Active Record.
- ActionDispatch::Session::CacheStore – Stores the data in the Rails cache.
- ActionDispatch::Session::MemCacheStore – Stores the data in a memcached cluster (this is a legacy implementation; consider using CacheStore instead).

All session stores use a cookie to store a unique ID for each session (you must use a cookie, Rails will not allow you to pass the session ID in the URL as this is less secure).

For most stores this ID is used to look up the session data on the server, e.g. in a database table. There is one exception, and that is the default and recommended session store – the CookieStore – which stores all session data in the cookie itself (the ID is still available to you if you need it). This has the advantage of being very lightweight and it requires zero setup in a new application in order to use the session. The cookie data is cryptographically signed to make it tamper-proof, but it is not encrypted, so anyone with access to it can read its contents but not edit it (Rails will not accept it if it has been edited).

The CookieStore can store around 4kB of data — much less than the others — but this is usually enough. Storing large amounts of data in the session is discouraged no matter which session store your application uses. You should especially avoid storing complex objects (anything other than basic Ruby objects, the most common example being model instances) in the session, as the server might not be able to reassemble them between requests, which will result in an error.

If your user sessions don't store critical data or don't need to be around for long periods (for instance if you just use the flash for messaging), you can consider using ActionDispatch::Session::CacheStore. This will store sessions using the cache implementation you have configured for your application. The advantage of this is that you can use your existing cache infrastructure for storing sessions without requiring any additional setup or administration. The downside, of course, is that the sessions will be ephemeral and could disappear at any time.

Read more about session storage in the Security Guide.

If you need a different session storage mechanism, you can change it in the
config/initializers/session_store.rb file:

```
# Use the database for sessions instead of the cookie-based default,
# which shouldn't be used to store highly confidential information
# (create the session table with "script/rails g session_migration")
# YourApp::Application.config.session_store :active_record_store
```

Rails sets up a session key (the name of the cookie) when signing the session data. These can also be changed in config/initializers/session_store.rb:

```
# Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store :cookie_store,
                                           :key => '_your_app_session'
```

You can also pass a :domain key and specify the domain name for the cookie:

```
# Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store :cookie_store,
:key => '_your_app_session', :domain => ".example.com"
```

Rails sets up (for the CookieStore) a secret key used for signing the session data. This can be changed in config/initializers/secret_token.rb

```
# Be sure to restart your server when you modify this file.

# Your secret key for verifying the integrity of signed cookies.
# If you change this key, all old signed cookies will become invalid!
# Make sure the secret is at least 30 characters and all random,
# no regular words or you'll be exposed to dictionary attacks.
YourApp::Application.config.secret_token = '49d3f3de9ed86c74b94ad6bd0...'
```

Changing the secret when using the CookieStore will invalidate all existing sessions.

8.4.1 Accessing the Session

In your controller you can access the session through the session instance method.

Sessions are lazily loaded. If you don't access sessions in your action's code, they will not be loaded. Hence you will never need to disable sessions, just not accessing them will do the job.

Session values are stored using key/value pairs like a hash:

```
class ApplicationController < ActionController::Base

  private

  # Finds the User with the ID stored in the session with the key
  # :current_user_id This is a common way to handle user login in
  # a Rails application; logging in sets the session value and
  # logging out removes it.
  def current_user
    @_current_user ||= session[:current_user_id] &&
      User.find_by_id(session[:current_user_id])
  end
end
```

To store something in the session, just assign it to the key like a hash:

```
class LoginsController < ApplicationController
  # "Create" a login, aka "log the user in"
  def create
    if user = User.authenticate(params[:username], params[:password])
      # Save the user ID in the session so it can be used in
      # subsequent requests
      session[:current_user_id] = user.id
      redirect_to root_url
    end
  end
end
```

To remove something from the session, assign that key to be nil:

```
class LoginsController < ApplicationController
  # "Delete" a login, aka "log the user out"
  def destroy
    # Remove the user id from the session
    @_current_user = session[:current_user_id] = nil
    redirect_to root_url
  end
end
```

To reset the entire session, use reset_session.

8.4.2 The Flash

The flash is a special part of the session which is cleared with each request. This means that values stored there will only be available in the next request, which is useful for storing error messages etc. It is accessed in much the same way as the session, like a hash. Let's use the act of logging out as an example. The controller can send a message which will be displayed to the user on the next request:

```
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged out"
    redirect_to root_url
  end
end
```

Note it is also possible to assign a flash message as part of the redirection.

```
redirect_to root_url, :notice => "You have successfully logged out"
```

The destroy action redirects to the application's root_url, where the message will be displayed. Note that it's entirely up to the next action to decide what, if anything, it will do with what the previous action put in the flash. It's conventional to display eventual errors or notices from the flash in the application's layout:

```
<html>
  <!-- <head/> -->
  <body>
    <% if flash[:notice] %>
      <p class="notice"><%= flash[:notice] %></p>
    <% end %>
    <% if flash[:error] %>
      <p class="error"><%= flash[:error] %></p>
    <% end %>
    <!-- more content -->
  </body>
</html>
```

This way, if an action sets an error or a notice message, the layout will display it automatically.

If you want a flash value to be carried over to another request, use the keep method:

```
class MainController < ApplicationController
  # Let’s say this action corresponds to root_url, but you want
  # all requests here to be redirected to UsersController#index.
  # If an action sets the flash and redirects here, the values
  # would normally be lost when another redirect happens, but you
  # can use ’keep’ to make it persist for another request.
  def index
    # Will persist all flash values.
    flash.keep

    # You can also use a key to keep only some kind of value.
    # flash.keep(:notice)
    redirect_to users_url
  end
end
```

flash.now

By default, adding values to the flash will make them available to the next request, but sometimes you may want to access those values in the same request. For example, if the create action fails to save a resource and you render the new template directly, that’s not going to result in a new request, but you may still want to display a message using the flash. To do this, you can use flash.now in the same way you use the normal flash:

```
class ClientsController < ApplicationController
  def create
    @client = Client.new(params[:client])
    if @client.save
      # ...
    else
      flash.now[:error] = "Could not save client"
      render :action => "new"
    end
  end
end
```

8.5 Cookies

Your application can store small amounts of data on the client — called cookies — that will be persisted across requests and even sessions. Rails provides easy access to cookies via the cookies method, which — much like the session — works like a hash:

```
class CommentsController < ApplicationController
  def new
    # Auto-fill the commenter's name if it has been stored in a cookie
    @comment = Comment.new(:name => cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(params[:comment])
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
        # Remember the commenter's name.
        cookies[:commenter_name] = @comment.name
      else
        # Delete cookie for the commenter's name cookie, if any.
        cookies.delete(:commenter_name)
      end
      redirect_to @comment.article
    else
      render :action => "new"
    end
  end
end
```

Note that while for session values you set the key to nil, to delete a cookie value you should use cookies.delete(:key).

8.6 Rendering xml and json data

ActionController makes it extremely easy to render xml or json data. If you generate a controller using scaffold then your controller would look something like this.

```
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @users}
      format.json { render :json => @users}
    end
  end
end
```

Notice that in the above case code is render :xml => @users and not render :xml => @users.to_xml. That is because if the input is not string then rails automatically invokes to_xml .

8.7 Filters

Filters are methods that are run before, after or "around" a controller action.

Filters are inherited, so if you set a filter on ApplicationController, it will be run on every controller in your application.

Before filters may halt the request cycle. A common before filter is one which requires that a user is logged in for an action to be run. You can define the filter method this way:

```
class ApplicationController < ActionController::Base
  before_filter :require_login

  private

  def require_login
```

```
    unless logged_in?
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url # halts request cycle
    end
  end

  # The logged_in? method simply returns true if the user is logged
  # in and false otherwise. It does this by "booleanizing" the
  # current_user method we created previously using a double ! operator.
  # Note that this is not common in Ruby and is discouraged unless you
  # really mean to convert something into true or false.
  def logged_in?
    !!current_user
  end
end
```

The method simply stores an error message in the flash and redirects to the login form if the user is not logged in. If a before filter renders or redirects, the action will not run. If there are additional filters scheduled to run after that filter they are also cancelled.

In this example the filter is added to ApplicationController and thus all controllers in the application inherit it. This will make everything in the application require the user to be logged in in order to use it. For obvious reasons (the user wouldn't be able to log in in the first place!), not all controllers or actions should require this. You can prevent this filter from running before particular actions with skip_before_filter:

```
class LoginsController < ApplicationController
  skip_before_filter :require_login, :only => [:new, :create]
end
```

Now, the LoginsController's new and create actions will work as before without requiring the user to be logged in. The :only option is used to only skip this filter for these actions, and there is also an :except option which works the other way. These options can be used when adding filters too, so you can add a filter which only runs for selected actions in the first place.

8.7.1 After Filters and Around Filters

In addition to before filters, you can also run filters after an action has been executed, or both before and after.

After filters are similar to before filters, but because the action has already been run they have access to the response data that's about to be sent to the client. Obviously, after filters cannot stop the action from running.

Around filters are responsible for running their associated actions by yielding, similar to how Rack middlewares work.

For example, in a website where changes have an approval workflow an administrator could be able to preview them easily, just apply them within a transaction:

```
class ChangesController < ActionController::Base
  around_filter :wrap_in_transaction, :only => :show

  private

  def wrap_in_transaction
    ActiveRecord::Base.transaction do
      begin
        yield
      ensure
        raise ActiveRecord::Rollback
      end
    end
  end
end
```

Note that an around filter wraps also rendering. In particular, if in the example above the view itself reads from the database via a scope or whatever, it will do so within the transaction and thus present the data to preview.

They can choose not to yield and build the response themselves, in which case the action is not run.

8.7.2 Other Ways to Use Filters

While the most common way to use filters is by creating private methods and using *_filter to add them, there are two other ways to do the same thing.

The first is to use a block directly with the *_filter methods. The block receives the controller as an argument, and the require_login filter from above could be rewritten to use a block:

```
class ApplicationController < ActionController::Base
  before_filter do |controller|
    redirect_to new_login_url unless controller.send(:logged_in?)
  end
end
```

Note that the filter in this case uses send because the logged_in? method is private and the filter is not run in the scope of the controller. This is not the recommended way to implement this particular filter, but in more simple cases it might be useful.

The second way is to use a class (actually, any object that responds to the right methods will do) to handle the filtering. This is useful in cases that are more complex and can not be implemented in a readable and reusable way using the two other methods. As an example, you could rewrite the login filter again to use a class:

```
class ApplicationController < ActionController::Base
  before_filter LoginFilter
end

class LoginFilter
  def self.filter(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in"
      controller.redirect_to controller.new_login_url
    end
  end
end
```

Again, this is not an ideal example for this filter, because it's not run in the scope of the controller but gets the controller passed as an argument. The filter class has a class method filter which gets run before or after the action, depending on if it's a before or after filter. Classes used as around filters can also use the same filter method, which will get run in the same way. The method must yield to execute the action. Alternatively, it can have both a before and an after method that are run before and after the action.

8.8 Request Forgery Protection

Cross-site request forgery is a type of attack in which a site tricks a user into making requests on another site, possibly adding, modifying or deleting data on that site without the user's knowledge or permission.

The first step to avoid this is to make sure all "destructive" actions (create, update and destroy) can only be accessed with non-GET requests. If you're following RESTful conventions you're already doing this. However, a malicious site can still send a non-GET request to your site quite easily, and that's where the request forgery protection comes in. As the name says, it protects from forged requests.

The way this is done is to add a non-guessable token which is only known to your server to each request. This way, if a request comes in without the proper token, it will be denied access.

If you generate a form like this:

```
<%= form_for @user do |f| %>
  <%= f.text_field :username %>
  <%= f.text_field :password %>
<% end %>
```

You will see how the token gets added as a hidden field:

```
<form accept-charset="UTF-8" action="/users/1" method="post">
<input type="hidden"
       value="67250ab105eb5ad10851c00a5621854a23af5489"
```

```
      name="authenticity_token"/>
<!-- fields -->
</form>
```

Rails adds this token to every form that's generated using the form helpers, so most of the time you don't have to worry about it. If you're writing a form manually or need to add the token for another reason, it's available through the method form_authenticity_token:

The form_authenticity_token generates a valid authentication token. That's useful in places where Rails does not add it automatically, like in custom Ajax calls.

The Security Guide has more about this and a lot of other security-related issues that you should be aware of when developing a web application.

8.9 The Request and Response Objects

In every controller there are two accessor methods pointing to the request and the response objects associated with the request cycle that is currently in execution. The request method contains an instance of AbstractRequest and the response method returns a response object representing what is going to be sent back to the client.

8.9.1 The request Object

The request object contains a lot of useful information about the request coming in from the client. To get a full list of the available methods, refer to the API documentation. Among the properties that you can access on this object are:

Property of request	Purpose
host	The hostname used for this request.
domain(n=2)	The hostname's first n segments, starting from the right (the TLD).
format	The content type requested by the client.
method	The HTTP method used for the request.
get?, post?, put?, delete?, head?	Returns true if the HTTP method is GET/POST/PUT/DELETE/HEAD.
headers	Returns a hash containing the headers associated with the request.
port	The port number (integer) used for the request.
protocol	Returns a string containing the protocol used plus "://", for example "http://".
query_string	The query string part of the URL, i.e., everything after "?".
remote_ip	The IP address of the client.
url	The entire URL used for the request.

path_parameters, query_parameters, and request_parameters

Rails collects all of the parameters sent along with the request in the params hash, whether they are sent as part of the query string or the post body. The request object has three accessors that give you access to these parameters depending on where they came from. The query_parameters hash contains parameters that were sent as part of the query string while the request_parameters hash contains parameters sent as part of the post body. The path_parameters hash contains parameters that were recognized by the routing as being part of the path leading to this particular controller and action.

8.9.2 The response Object

The response object is not usually used directly, but is built up during the execution of the action and rendering of the data that is being sent back to the user, but sometimes – like in an after filter – it can be useful to access the response directly. Some of these accessor methods also have setters, allowing you to change their values.

Property of response	Purpose
body	This is the string of data being sent back to the client. This is most often HTML.
status	The HTTP status code for the response, like 200 for a successful request or 404 for file not found.
location	The URL the client is being redirected to, if any.
content_type	The content type of the response.
charset	The character set being used for the response. Default is "utf-8".
headers	Headers used for the response.

Setting Custom Headers

If you want to set custom headers for a response then response.headers is the place to do it. The headers attribute is a hash which maps header names to their values, and Rails will set some of them automatically. If you want to add or change a header, just assign it to response.headers this way:

```
response.headers["Content-Type"] = "application/pdf"
```

8.10 HTTP Authentications

Rails comes with two built-in HTTP authentication mechanisms:

- Basic Authentication
- Digest Authentication

8.10.1 HTTP Basic Authentication

HTTP basic authentication is an authentication scheme that is supported by the majority of browsers and other HTTP clients. As an example, consider an administration section which will only be available by entering a username and a password into the browser's HTTP basic dialog window. Using the built-in authentication is quite easy and only requires you to use one method, http_basic_authenticate_with.

```
class AdminController < ApplicationController
  http_basic_authenticate_with :name => "humbaba", :password => "5baa61e4"
end
```

With this in place, you can create namespaced controllers that inherit from AdminController. The filter will thus be run for all actions in those controllers, protecting them with HTTP basic authentication.

8.10.2 HTTP Digest Authentication

HTTP digest authentication is superior to the basic authentication as it does not require the client to send an unencrypted password over the network (though HTTP basic authentication is safe over HTTPS). Using digest authentication with Rails is quite easy and only requires using one method, authenticate_or_request_with_http_digest.

```
class AdminController < ApplicationController
  USERS = { "lifo" => "world" }

  before_filter :authenticate

  private
  def authenticate
    authenticate_or_request_with_http_digest do |username|
      USERS[username]
    end
  end
end
```

As seen in the example above, the authenticate_or_request_with_http_digest block takes only one argument – the username. And the block returns the password. Returning false or nil from the authenticate_or_request_with_http_digest will cause authentication failure.

8.11 Streaming and File Downloads

Sometimes you may want to send a file to the user instead of rendering an HTML page. All controllers in Rails have the send_data and the send_file methods, which will both stream data to the client. send_file is a convenience method that lets you provide the name of a file on the disk and it will stream the contents of that file for you.

To stream data to the client, use send_data:

```
require "prawn"
class ClientsController < ApplicationController
  # Generates a PDF document with information on the client and
  # returns it. The user will get the PDF as a file download.
  def download_pdf
    client = Client.find(params[:id])
    send_data generate_pdf(client),
              :filename => "#{client.name}.pdf",
              :type => "application/pdf"
  end

  private

  def generate_pdf(client)
    Prawn::Document.new do
```

```
      text client.name, :align => :center
      text "Address: #{client.address}"
      text "Email: #{client.email}"
    end.render
  end
end
```

The download_pdf action in the example above will call a private method which actually generates the PDF document and returns it as a string. This string will then be streamed to the client as a file download and a filename will be suggested to the user. Sometimes when streaming files to the user, you may not want them to download the file. Take images, for example, which can be embedded into HTML pages. To tell the browser a file is not meant to be downloaded, you can set the :disposition option to "inline". The opposite and default value for this option is "attachment".

8.11.1 Sending Files

If you want to send a file that already exists on disk, use the send_file method.

```
class ClientsController < ApplicationController
  # Stream a file that has already been generated and stored on disk.
  def download_pdf
    client = Client.find(params[:id])
    send_file("#{Rails.root}/files/clients/#{client.id}.pdf",
              :filename => "#{client.name}.pdf",
              :type => "application/pdf")
  end
end
```

This will read and stream the file 4kB at the time, avoiding loading the entire file into memory at once. You can turn off streaming with the :stream option or adjust the block size with the :buffer_size option.

If :type is not specified, it will be guessed from the file extension specified in :filename. If the content type is not registered for the extension, application/octet-stream will be used.

Be careful when using data coming from the client (params, cookies, etc.) to locate the file on disk, as this is a security risk that might allow someone to gain access to files they are not meant to see.

It is not recommended that you stream static files through Rails if you can instead keep them in a public folder on your web server. It is much more efficient to let the user download the file directly using Apache or another web server, keeping the request from unnecessarily going through the whole Rails stack.

8.11.2 RESTful Downloads

While send_data works just fine, if you are creating a RESTful application having separate actions for file downloads is usually not necessary. In REST terminology, the PDF file from the example above can be considered just another representation of the client resource. Rails provides an easy and quite sleek way of doing "RESTful downloads". Here's how you can rewrite the example so that the PDF download is a part of the show action, without any streaming:

```
class ClientsController < ApplicationController
  # The user can request to receive this resource as HTML or PDF.
  def show
    @client = Client.find(params[:id])

    respond_to do |format|
      format.html
      format.pdf { render :pdf => generate_pdf(@client) }
    end
  end
end
```

In order for this example to work, you have to add the PDFMIME type to Rails. This can be done by adding the following line to the file config/initializers/mime_types.rb:

```
Mime::Type.register "application/pdf", :pdf
```

Configuration files are not reloaded on each request, so you have to restart the server in order for their changes to take effect.

Now the user can request to get a PDF version of a client just by adding ".pdf" to the URL:

```
GET /clients/1.pdf
```

8.12 Parameter Filtering

Rails keeps a log file for each environment in the log folder. These are extremely useful when debugging what's actually going on in your application, but in a live application you may not want every bit of information to be stored in the log file. You can filter certain request parameters from your log files by appending them to config.filter_parameters in the application configuration. These parameters will be marked [FILTERED] in the log.

```
config.filter_parameters << :password
```

8.13 Rescue

Most likely your application is going to contain bugs or otherwise throw an exception that needs to be handled. For example, if the user follows a link to a resource that no longer exists in the database, Active Record will throw the ActiveRecord::RecordNotFound exception.

Rails' default exception handling displays a "500 Server Error" message for all exceptions. If the request was made locally, a nice traceback and some added information gets displayed so you can figure out what went wrong and deal with it. If the request was remote Rails will just display a simple "500 Server Error" message to the user, or a "404 Not Found" if there was a routing error or a record could not be found. Sometimes you might want to customize how these errors are caught and how they're displayed to the user. There are several levels of exception handling available in a Rails application:

8.13.1 The Default 500 and 404 Templates

By default a production application will render either a 404 or a 500 error message. These messages are contained in static HTML files in the public folder, in 404.html and 500.html respectively. You can customize these files to add some extra information and layout, but remember that they are static; i.e. you can't use RHTML or layouts in them, just plain HTML.

8.13.2 rescue_from

If you want to do something a bit more elaborate when catching errors, you can use rescue_from, which handles exceptions of a certain type (or multiple types) in an entire controller and its subclasses.

When an exception occurs which is caught by a rescue_from directive, the exception object is passed to the handler. The handler can be a method or a Proc object passed to the :with option. You can also use a block directly instead of an explicit Proc object.

Here's how you can use rescue_from to intercept all ActiveRecord::RecordNotFound errors and do something with them.

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound, :with => :record_not_found

  private

  def record_not_found
    render :text => "404 Not Found", :status => 404
  end
end
```

Of course, this example is anything but elaborate and doesn't improve on the default exception handling at all, but once you can catch all those exceptions you're free to do whatever you want with them. For example, you could create custom exception classes that will be thrown when a user doesn't have access to a certain section of your application:

```
class ApplicationController < ActionController::Base
  rescue_from User::NotAuthorized, :with => :user_not_authorized

  private

  def user_not_authorized
    flash[:error] = "You don't have access to this section."
    redirect_to :back
  end
end

class ClientsController < ApplicationController
  # Check that the user has the right authorization to access clients.
  before_filter :check_authorization

  # Note how the actions don't have to worry about all the auth stuff.
  def edit
    @client = Client.find(params[:id])
  end

  private

  # If the user is not authorized, just throw the exception.
  def check_authorization
    raise User::NotAuthorized unless current_user.admin?
  end
end
```

Certain exceptions are only rescuable from the ApplicationController class, as they are raised before the controller gets initialized and the action gets executed. See Pratik Naik's article on the subject for more information.

8.14 Force HTTPS protocol

Sometime you might want to force a particular controller to only be accessible via an HTTPS protocol for security reasons. Since Rails 3.1 you can now use force_ssl method in your controller to enforce that:

```
class DinnerController
  force_ssl
end
```

Just like the filter, you could also passing :only and :except to enforce the secure connection only to specific actions.

```
class DinnerController
  force_ssl :only => :cheeseburger
  # or
  force_ssl :except => :cheeseburger
end
```

Please note that if you found yourself adding force_ssl to many controllers, you may found yourself wanting to force the whole application to use HTTPS instead. In that case, you can set the config.force_ssl in your environment file.

Chapter 9

Rails Routing from the Outside In

This guide covers the user-facing features of Rails routing. By referring to this guide, you will be able to:

- Understand the code in routes.rb
- Construct your own routes, using either the preferred resourceful style or the match method
- Identify what parameters to expect an action to receive
- Automatically create paths and URLs using route helpers
- Use advanced techniques such as constraints and Rack endpoints

9.1 The Purpose of the Rails Router

The Rails router recognizes URLs and dispatches them to a controller's action. It can also generate paths and URLs, avoiding the need to hardcode strings in your views.

9.1.1 Connecting URLs to Code

When your Rails application receives an incoming request

```
GET /patients/17
```

it asks the router to match it to a controller action. If the first matching route is

```
match "/patients/:id" => "patients#show"
```

the request is dispatched to the patients controller's show action with { :id => "17" } in params.

9.1.2 Generating Paths and URLs from Code

You can also generate paths and URLs. If your application contains this code:

```
@patient = Patient.find(17)
```

```
<%= link_to "Patient Record", patient_path(@patient) %>
```

The router will generate the path /patients/17. This reduces the brittleness of your view and makes your code easier to understand. Note that the id does not need to be specified in the route helper.

9.2 Resource Routing: the Rails Default

Resource routing allows you to quickly declare all of the common routes for a given resourceful controller. Instead of declaring separate routes for your index, show, new, edit, create, update and destroy actions, a resourceful route declares them in a single line of code.

9.2.1 Resources on the Web

Browsers request pages from Rails by making a request for a URL using a specific HTTP method, such as GET, POST, PUT and DELETE. Each method is a request to perform an operation on the resource. A resource route maps a number of related requests to actions in a single controller.

When your Rails application receives an incoming request for

```
DELETE /photos/17
```

it asks the router to map it to a controller action. If the first matching route is

```
resources :photos
```

Rails would dispatch that request to the destroy method on the photos controller with { :id => "17" } in params.

9.2.2 CRUD, Verbs, and Actions

In Rails, a resourceful route provides a mapping between HTTP verbs and URLs to controller actions. By convention, each action also maps to particular CRUD operations in a database. A single entry in the routing file, such as

```
resources :photos
```

creates seven different routes in your application, all mapping to the Photos controller:

Verb	**Path**	**action**	**used for**
GET	/photos	index	display a list of all photos
GET	/photos/new	new	return an HTML form for creating a new photo
POST	/photos	create	create a new photo
GET	/photos/:id	show	display a specific photo
GET	/photos/:id/edit	edit	return an HTML form for editing a photo
PUT	/photos/:id	update	update a specific photo
DELETE	/photos/:id	destroy	delete a specific photo

Rails routes are matched in the order they are specified, so if you have a resources :photos above a get 'photos/poll' the show action's route for the resources line will be matched before the get line. To fix this, move the get line **above** the resources line so that it is matched first.

9.2.3 Paths and URLs

Creating a resourceful route will also expose a number of helpers to the controllers in your application. In the case of resources :photos:

- photos_path returns /photos
- new_photo_path returns /photos/new
- edit_photo_path(:id) returns /photos/:id/edit
  (for instance, edit_photo_path(10) returns /photos/10/edit)
- photo_path(:id) returns /photos/:id (for instance, photo_path(10) returns /photos/10)

Each of these helpers has a corresponding _url helper (such as photos_url) which returns the same path prefixed with the current host, port and path prefix.

Because the router uses the HTTP verb and URL to match inbound requests, four URLs map to seven different actions.

9.2.4 Defining Multiple Resources at the Same Time

If you need to create routes for more than one resource, you can save a bit of typing by defining them all with a single call to resources:

```
resources :photos, :books, :videos
```

This works exactly the same as

```
resources :photos
resources :books
resources :videos
```

9.2.5 Singular Resources

Sometimes, you have a resource that clients always look up without referencing an ID. For example, you would like /profile to always show the profile of the currently logged in user. In this case, you can use a singular resource to map /profile (rather than /profile/:id) to the show action.

```
match "profile" => "users#show"
```

This resourceful route

```
resource :geocoder
```

creates six different routes in your application, all mapping to the Geocoders controller:

Verb	**Path**	**action**	**used for**
GET	/geocoder/new	new	return an HTML form for creating the geocoder
POST	/geocoder	create	create the new geocoder
GET	/geocoder	show	display the one and only geocoder resource
GET	/geocoder/edit	edit	return an HTML form for editing the geocoder
PUT	/geocoder	update	update the one and only geocoder resource
DELETE	/geocoder	destroy	delete the geocoder resource

Because you might want to use the same controller for a singular route (/account) and a plural route (/accounts/45), singular resources map to plural controllers.

A singular resourceful route generates these helpers:

- new_geocoder_path returns /geocoder/new
- edit_geocoder_path returns /geocoder/edit
- geocoder_path returns /geocoder

As with plural resources, the same helpers ending in _url will also include the host, port and path prefix.

9.2.6 Controller Namespaces and Routing

You may wish to organize groups of controllers under a namespace. Most commonly, you might group a number of administrative controllers under an Admin:: namespace. You would place these controllers under the app/controllers/admin directory, and you can group them together in your router:

```
namespace :admin do
  resources :posts, :comments
end
```

This will create a number of routes for each of the posts and comments controller. For Admin::PostsController, Rails will create:

Verb	Path	action	named helper
GET	/admin/posts	index	admin_posts_path
GET	/admin/posts/new	new	new_admin_post_path
POST	/admin/posts	create	admin_posts_path
GET	/admin/posts/:id	show	admin_post_path(:id)
GET	/admin/posts/:id/edit	edit	edit_admin_post_path(:id)
PUT	/admin/posts/:id	update	admin_post_path(:id)
DELETE	/admin/posts/:id	destroy	admin_post_path(:id)

If you want to route /posts (without the prefix /admin) to Admin::PostsController, you could use

```
scope :module => "admin" do
  resources :posts, :comments
end
```

or, for a single case

```
resources :posts, :module => "admin"
```

If you want to route /admin/posts to PostsController (without the Admin:: module prefix), you could use

```
scope "/admin" do
  resources :posts, :comments
end
```

or, for a single case

```
resources :posts, :path => "/admin/posts"
```

In each of these cases, the named routes remain the same as if you did not use scope. In the last case, the following paths map to PostsController:

Verb	Path	action	named helper
GET	/admin/posts	index	posts_path
GET	/admin/posts/new	new	new_post_path
POST	/admin/posts	create	posts_path
GET	/admin/posts/:id	show	post_path(:id)
GET	/admin/posts/:id/edit	edit	edit_post_path(:id)
PUT	/admin/posts/:id	update	post_path(:id)
DELETE	/admin/posts/:id	destroy	post_path(:id)

9.2.7 Nested Resources

It's common to have resources that are logically children of other resources. For example, suppose your application includes these models:

```
class Magazine < ActiveRecord::Base
  has_many :ads
end

class Ad < ActiveRecord::Base
  belongs_to :magazine
end
```

Nested routes allow you to capture this relationship in your routing. In this case, you could include this route declaration:

```
resources :magazines do
  resources :ads
end
```

In addition to the routes for magazines, this declaration will also route ads to an AdsController. The ad URLs require a magazine:

Verb	Path	action	used for
GET	/magazines /:magazine_id /ads	index	display a list of all ads for a specific magazine
GET	/magazines /:magazine_id /ads /new	new	return an HTML form for creating a new ad belonging to a specific magazine
POST	/magazines /:magazine_id /ads	create	create a new ad belonging to a specific magazine
GET	/magazines /:magazine_id /ads /:id	show	display a specific ad belonging to a specific magazine
GET	/magazines /:magazine_id /ads /:id /edit	edit	return an HTML form for editing an ad belonging to a specific magazine
PUT	/magazines /:magazine_id /ads /:id	update	update a specific ad belonging to a specific magazine
DELETE	/magazines /:magazine_id /ads /:id	destroy	delete a specific ad belonging to a specific magazine

This will also create routing helpers such as magazine_ads_url and edit_magazine_ad_path. These helpers take an instance of Magazine as the first parameter (magazine_ads_url(@magazine)).

Limits to Nesting

You can nest resources within other nested resources if you like. For example:

```
resources :publishers do
  resources :magazines do
    resources :photos
  end
end
```

Deeply-nested resources quickly become cumbersome. In this case, for example, the application would recognize paths such as

```
/publishers/1/magazines/2/photos/3
```

The corresponding route helper would be publisher_magazine_photo_url, requiring you to specify objects at all three levels. Indeed, this situation is confusing enough that a popular article by Jamis Buck proposes a rule of thumb for good Rails design:

Resources should never be nested more than 1 level deep.

9.2.8 Creating Paths and URLs From Objects

In addition to using the routing helpers, Rails can also create paths and URLs from an array of parameters. For example, suppose you have this set of routes:

```
resources :magazines do
  resources :ads
end
```

When using magazine_ad_path, you can pass in instances of Magazine and Ad instead of the numeric IDs.

```
<%= link_to "Ad details", magazine_ad_path(@magazine, @ad) %>
```

You can also use url_for with a set of objects, and Rails will automatically determine which route you want:

```
<%= link_to "Ad details", url_for([@magazine, @ad]) %>
```

In this case, Rails will see that @magazine is a Magazine and @ad is an Ad and will therefore use the magazine_ad_path helper. In helpers like link_to, you can specify just the object in place of the full url_for call:

```
<%= link_to "Ad details", [@magazine, @ad] %>
```

If you wanted to link to just a magazine, you could leave out the Array:

```
<%= link_to "Magazine details", @magazine %>
```

This allows you to treat instances of your models as URLs, and is a key advantage to using the resourceful style.

9.2.9 Adding More RESTful Actions

You are not limited to the seven routes that RESTful routing creates by default. If you like, you may add additional routes that apply to the collection or individual members of the collection.

Adding Member Routes

To add a member route, just add a member block into the resource block:

```
resources :photos do
  member do
    get 'preview'
  end
end
```

This will recognize /photos/1/preview with GET, and route to the preview action of PhotosController. It will also create the preview_photo_url and preview_photo_path helpers.

Within the block of member routes, each route name specifies the HTTP verb that it will recognize. You can use get, put, post, or delete here. If you don't have multiple member routes, you can also pass :on to a route, eliminating the block:

```
resources :photos do
  get 'preview', :on => :member
end
```

Adding Collection Routes

To add a route to the collection:

```
resources :photos do
  collection do
    get 'search'
  end
end
```

This will enable Rails to recognize paths such as /photos/search with GET, and route to the search action of PhotosController. It will also create the search_photos_url and search_photos_path route helpers.

Just as with member routes, you can pass :on to a route:

```
resources :photos do
  get 'search', :on => :collection
end
```

A Note of Caution

If you find yourself adding many extra actions to a resourceful route, it's time to stop and ask yourself whether you're disguising the presence of another resource.

9.3 Non-Resourceful Routes

In addition to resource routing, Rails has powerful support for routing arbitrary URLs to actions. Here, you don't get groups of routes automatically generated by resourceful routing. Instead, you set up each route within your application separately.

While you should usually use resourceful routing, there are still many places where the simpler routing is more appropriate. There's no need to try to shoehorn every last piece of your application into a resourceful framework if that's not a good fit.

In particular, simple routing makes it very easy to map legacy URLs to new Rails actions.

9.3.1 Bound Parameters

When you set up a regular route, you supply a series of symbols that Rails maps to parts of an incoming HTTP request. Two of these symbols are special: :controller maps to the name of a controller in your application, and :action maps to the name of an action within that controller. For example, consider one of the default Rails routes:

```
match ':controller(/:action(/:id))'
```

If an incoming request of /photos/show/1 is processed by this route (because it hasn't matched any previous route in the file), then the result will be to invoke the show action of the PhotosController, and to make the final parameter "1" available as params[:id]. This route will also route the incoming request of /photos to PhotosController#index, since :action and :id are optional parameters, denoted by parentheses.

9.3.2 Dynamic Segments

You can set up as many dynamic segments within a regular route as you like. Anything other than :controller or :action will be available to the action as part of params. If you set up this route:

```
match ':controller/:action/:id/:user_id'
```

An incoming path of /photos/show/1/2 will be dispatched to the show action of the PhotosController. params[:id] will be "1", and params[:user_id] will be "2".

You can't use :namespace or :module with a :controller path segment. If you need to do this then use a constraint on :controller that matches the namespace you require. e.g:

```
match ':controller(/:action(/:id))',
      :controller => /admin\/[^\/]+/
```

By default dynamic segments don't accept dots – this is because the dot is used as a separator for formatted routes. If you need to use a dot within a dynamic segment add a constraint which overrides this – for example :id => /[^\/]+/ allows anything except a slash.

9.3.3 Static Segments

You can specify static segments when creating a route:

```
match ':controller/:action/:id/with_user/:user_id'
```

This route would respond to paths such as /photos/show/1/with_user/2. In this case, params would be { :controller => "photos", :action => "show", :id => "1", :user_id => "2" }.

9.3.4 The Query String

The params will also include any parameters from the query string. For example, with this route:

```
match ':controller/:action/:id'
```

An incoming path of /photos/show/1?user_id=2 will be dispatched to the show action of the Photos controller. params will be { :controller => "photos", :action => "show", :id => "1", :user_id => "2" }.

9.3.5 Defining Defaults

You do not need to explicitly use the :controller and :action symbols within a route. You can supply them as defaults:

```
match 'photos/:id' => 'photos#show'
```

With this route, Rails will match an incoming path of /photos/12 to the show action of PhotosController.

You can also define other defaults in a route by supplying a hash for the :defaults option. This even applies to parameters that you do not specify as dynamic segments. For example:

```
match 'photos/:id' => 'photos#show',
      :defaults => { :format => 'jpg' }
```

Rails would match photos/12 to the show action of PhotosController, and set params[:format] to "jpg".

9.3.6 Naming Routes

You can specify a name for any route using the :as option.

```
match 'exit' => 'sessions#destroy', :as => :logout
```

This will create logout_path and logout_url as named helpers in your application. Calling logout_path will return /exit

9.3.7 HTTP Verb Constraints

You can use the :via option to constrain the request to one or more HTTP methods:

```
match 'photos/show' => 'photos#show', :via => :get
```

There is a shorthand version of this as well:

```
get 'photos/show'
```

You can also permit more than one verb to a single route:

```
match 'photos/show' => 'photos#show', :via => [:get, :post]
```

9.3.8 Segment Constraints

You can use the :constraints option to enforce a format for a dynamic segment:

```
match 'photos/:id' => 'photos#show',
:constraints => { :id => /[A-Z]\d{5}/ }
```

This route would match paths such as /photos/A12345. You can more succinctly express the same route this way:

```
match 'photos/:id' => 'photos#show', :id => /[A-Z]\d{5}/
```

:constraints takes regular expressions with the restriction that regexp anchors can't be used. For example, the following route will not work:

```
match '/:id' => 'posts#show', :constraints => {:id => /^\d/}
```

However, note that you don't need to use anchors because all routes are anchored at the start.

For example, the following routes would allow for posts with to_param values like 1-hello-world that always begin with a number and users with to_param values like david that never begin with a number to share the root namespace:

```
match '/:id' => 'posts#show', :constraints => { :id => /\d.+/ }
match '/:username' => 'users#show'
```

9.3.9 Request-Based Constraints

You can also constrain a route based on any method on the Request object that returns a String.

You specify a request-based constraint the same way that you specify a segment constraint:

```
match "photos", :constraints => {:subdomain => "admin"}
```

You can also specify constraints in a block form:

```
namespace :admin do
  constraints :subdomain => "admin" do
    resources :photos
  end
end
```

9.3.10 Advanced Constraints

If you have a more advanced constraint, you can provide an object that responds to matches? that Rails should use. Let's say you wanted to route all users on a blacklist to the BlacklistController. You could do:

```
class BlacklistConstraint
  def initialize
    @ips = Blacklist.retrieve_ips
  end

  def matches?(request)
    @ips.include?(request.remote_ip)
  end
end

TwitterClone::Application.routes.draw do
  match "*path" => "blacklist#index",
    :constraints => BlacklistConstraint.new
end
```

9.3.11 Route Globbing

Route globbing is a way to specify that a particular parameter should be matched to all the remaining parts of a route. For example

```
match 'photos/*other' => 'photos#unknown'
```

This route would match photos/12 or /photos/long/path/to/12, setting params[:other] to "12" or "long/path/to/12".
Wildcard segments can occur anywhere in a route. For example,

```
match 'books/*section/:title' => 'books#show'
```

would match books/some/section/last-words-a-memoir with params[:sec
equals "some/section", and params[:title] equals "last-words-a-memoir".
Technically a route can have even more than one wildcard segment. The matcher assigns segments to parameters in an intuitive way. For example,

```
match '*a/foo/*b' => 'test#index'
```

would match zoo/woo/foo/bar/baz with params[:a] equals "zoo/woo", and params[:b] equals "bar/baz".

Starting from Rails 3.1, wildcard routes will always match the optional format segment by default. For example if you have this route:

```
match '*pages' => 'pages#show'
```

By requesting "/foo/bar.json", your params[:pages] will be equals to "foo/bar" with the request format of JSON. If you want the old 3.0.x behavior back, you could supply :format => false like this:

```
match '*pages' => 'pages#show', :format => false
```

If you want to make the format segment mandatory, so it cannot be omitted, you can supply :format => true like this:

```
match '*pages' => 'pages#show', :format => true
```

9.3.12 Redirection

You can redirect any path to another path using the redirect helper in your router:

```
match "/stories" => redirect("/posts")
```

You can also reuse dynamic segments from the match in the path to redirect to:

```
match "/stories/:name" => redirect("/posts/%{name}")
```

You can also provide a block to redirect, which receives the params and (optionally) the request object:

```
match "/stories/:name"
=> redirect {|params| "/posts/#{params[:name].pluralize}" }

match "/stories"
=> redirect {|p, req| "/posts/#{req.subdomain}" }
```

Please note that this redirection is a 301 "Moved Permanently" redirect. Keep in mind that some web browsers or proxy servers will cache this type of redirect, making the old page inaccessible.

In all of these cases, if you don't provide the leading host (http://www.ex
Rails will take those details from the current request.

9.3.13 Routing to Rack Applications

Instead of a String, like "posts#index", which corresponds to the index action in the PostsController, you can specify any Rack application as the endpoint for a matcher.

```
match "/application.js" => Sprockets
```

As long as Sprockets responds to call and returns a [status, headers, body], the router won't know the difference between the Rack application and an action.

For the curious, "posts#index" actually expands out to PostsController.action(:index), which returns a valid Rack application.

9.3.14 Using root

You can specify what Rails should route "/" to with the root method:

```
root :to => 'pages#main'
```

You should put the root route at the top of the file, because it is the most popular route and should be matched first. You also need to delete the public/index.html file for the root route to take effect.

9.4 Customizing Resourceful Routes

While the default routes and helpers generated by resources :posts will usually serve you well, you may want to customize them in some way. Rails allows you to customize virtually any generic part of the resourceful helpers.

9.4.1 Specifying a Controller to Use

The :controller option lets you explicitly specify a controller to use for the resource. For example:

```
resources :photos, :controller => "images"
```

will recognize incoming paths beginning with /photos but route to the Images controller:

Verb	**Path**	**action**	**named helper**
GET	/photos	index	photos_path
GET	/photos/new	new	new_photo_path
POST	/photos	create	photos_path
GET	/photos/:id	show	photo_path(:id)
GET	/photos/:id/edit	edit	edit_photo_path(:id)
PUT	/photos/:id	update	photo_path(:id)
DELETE	/photos/:id	destroy	photo_path(:id)

Use photos_path, new_photo_path, etc. to generate paths for this resource.

9.4.2 Specifying Constraints

You can use the :constraints option to specify a required format on the implicit id. For example:

```
resources :photos, :constraints => {:id => /[A-Z][A-Z][0-9]+/}
```

This declaration constrains the :id parameter to match the supplied regular expression. So, in this case, the router would no longer match /photos/1 to this route. Instead, /photos/RR27 would match.

You can specify a single constraint to apply to a number of routes by using the block form:

```
constraints(:id => /[A-Z][A-Z][0-9]+/) do
  resources :photos
  resources :accounts
end
```

Of course, you can use the more advanced constraints available in non-resourceful routes in this context.

By default the :id parameter doesn't accept dots – this is because the dot is used as a separator for formatted routes. If you need to use a dot within an :id add a constraint which overrides this – for example :id => /[ˆ\/]+/ allows anything except a slash.

9.4.3 Overriding the Named Helpers

The :as option lets you override the normal naming for the named route helpers. For example:

```
resources :photos, :as => "images"
```

will recognize incoming paths beginning with /photos and route the requests to PhotosController, but use the value of the :as option to name the helpers.

Verb	**Path**	**action**	**named helper**
GET	/photos	index	images_path
GET	/photos/new	new	new_image_path
POST	/photos	create	images_path
GET	/photos/:id	show	image_path(:id)
GET	/photos/:id/edit	edit	edit_image_path(:id)
PUT	/photos/:id	update	image_path(:id)
DELETE	/photos/:id	destroy	image_path(:id)

9.4.4 Overriding the new and edit Segments

The :path_names option lets you override the automatically-generated "new" and "edit" segments in paths:

```
resources :photos, :path_names=>{:new=>'make',:edit=>'change'}
```

This would cause the routing to recognize paths such as

```
/photos/make
/photos/1/change
```

The actual action names aren't changed by this option. The two paths shown would still route to the new and edit actions.

If you find yourself wanting to change this option uniformly for all of your routes, you can use a scope.

```
scope :path_names => { :new => "make" } do
  # rest of your routes
end
```

9.4.5 Prefixing the Named Route Helpers

You can use the :as option to prefix the named route helpers that Rails generates for a route. Use this option to prevent name collisions between routes using a path scope.

```
scope "admin" do
  resources :photos, :as => "admin_photos"
end

resources :photos
```

This will provide route helpers such as admin_photos_path, new_admin_p etc.

To prefix a group of route helpers, use :as with scope:

```
scope "admin", :as => "admin" do
  resources :photos, :accounts
end

resources :photos, :accounts
```

This will generate routes such as admin_photos_path and admin_accounts_path which map to /admin/photos and /admin/accounts respectively.

The namespace scope will automatically add :as as well as :module and :path prefixes.

You can prefix routes with a named parameter also:

```
scope ":username" do
  resources :posts
end
```

This will provide you with URLs such as /bob/posts/1 and will allow you to reference the username part of the path as params[:username] in controllers, helpers and views.

9.4.6 Restricting the Routes Created

By default, Rails creates routes for the seven default actions (index, show, new, create, edit, update, and destroy) for every RESTful route in your application. You can use the :only and :except options to fine-tune this behavior. The :only option tells Rails to create only the specified routes:

```
resources :photos, :only => [:index, :show]
```

Now, a GET request to /photos would succeed, but a POST request to /photos (which would ordinarily be routed to the create action) will fail.

The :except option specifies a route or list of routes that Rails should not create:

```
resources :photos, :except => :destroy
```

In this case, Rails will create all of the normal routes except the route for destroy (a DELETE request to /photos/:id).

If your application has many RESTful routes, using :only and :except to generate only the routes that you actually need can cut down on memory use and speed up the routing process.

9.4.7 Translated Paths

Using scope, we can alter path names generated by resources:

```
scope(:path_names=>{:new=>"neu",:edit=>"bearbeiten"}) do
  resources :categories, :path => "kategorien"
end
```

Rails now creates routes to the CategoriesController.

Verb	Path	action	named helper
GET	/kategorien	index	categories_path
GET	/kategorien/neu	new	new_category_path
POST	/kategorien	create	categories_path
GET	/kategorien/:id	show	category_path(:id)
GET	/kategorien/:id/bearbeiten	edit	edit_category_path(:id)
PUT	/kategorien/:id	update	category_path(:id)
DELETE	/kategorien/:id	destroy	category_path(:id)

9.4.8 Overriding the Singular Form

If you want to define the singular form of a resource, you should add additional rules to the Inflector.

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular 'tooth', 'teeth'
end
```

9.4.9 Using :as in Nested Resources

The :as option overrides the automatically-generated name for the resource in nested route helpers. For example,

```
resources :magazines do
  resources :ads, :as => 'periodical_ads'
end
```

This will create routing helpers such as magazine_periodical_ads_url and edit_magazine_periodical_ad_path.

9.5 Inspecting and Testing Routes

Rails offers facilities for inspecting and testing your routes.

9.5.1 Seeing Existing Routes with rake

If you want a complete list of all of the available routes in your application, run rake routes command. This will print all of your routes, in the same order that they appear in routes.rb. For each route, you'll see:

- The route name (if any)
- The HTTP verb used (if the route doesn't respond to all verbs)
- The URL pattern to match
- The routing parameters for the route

For example, here's a small section of the rake routes output for a RESTful route:

```
    users  GET   /users(.:format)              users#index
           POST  /users(.:format)              users#create
 new_user  GET   /users/new(.:format)          users#new
edit_user  GET   /users/:id/edit(.:format)     users#edit
```

You may restrict the listing to the routes that map to a particular controller setting the CONTROLLER environment variable:

```
CONTROLLER=users rake routes
```

You'll find that the output from rake routes is much more readable if you widen your terminal window until the output lines don't wrap.

9.5.2 Testing Routes

Routes should be included in your testing strategy (just like the rest of your application). Rails offers three built-in assertions designed to make testing routes simpler:

- assert_generates
- assert_recognizes
- assert_routing

The assert_generates Assertion

assert_generates asserts that a particular set of options generate a particular path and can be used with default routes or custom routes.

```
assert_generates "/photos/1",
{ :controller => "photos", :action => "show", :id => "1" }

assert_generates "/about",
:controller => "pages", :action => "about"
```

The assert_recognizes Assertion

assert_recognizes is the inverse of assert_generates. It asserts that a given path is recognized and routes it to a particular spot in your application.

```
assert_recognizes({ :controller => "photos",
      :action => "show", :id => "1" }, "/photos/1")
```

You can supply a :method argument to specify the HTTP verb:

```
assert_recognizes({ :controller => "photos",
:action => "create" }, { :path => "photos", :method => :post })
```

The assert_routing Assertion

The assert_routing assertion checks the route both ways: it tests that the path generates the options, and that the options generate the path. Thus, it combines the functions of assert_generates and assert_recognizes.

```
assert_routing({ :path => "photos", :method => :post },
{ :controller => "photos", :action => "create" })
```

Chapter 10

License

Creative Commons Legal Code

Attribution-ShareAlike 3.0 Unported

License

OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

10.1 Definitions

1. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.
2. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other con-

tributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

3. **"Creative Commons Compatible License"** means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

4. **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

5. **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

6. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

7. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a per-

formance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

8. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

9. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

10. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place in-

dividually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

11. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

10.2 Fair Dealing Rights.

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

10.3 License Grant.

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

1. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
2. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was

translated from English to Spanish," or a modification could indicate "The original work has been modified.";

3. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

4. to Distribute and Publicly Perform Adaptations.

5. For the avoidance of doubt:

   (a) **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   (b) **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

   (c) **Voluntary License Schemes**. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

10.4 Restrictions.

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

2. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Ele-

bution (”Attribution Parties”) in Licensor’s copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., ”French translation of the Work by Original Author,” or ”Screenplay based on original Work by Original Author”). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

4. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author’s honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of

10.5 Representations, Warranties and Disclaimer

10.6 Limitation on Liability.

IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

10.7 Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

10.8 Miscellaneous

1. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
2. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work

10.9 Creative Commons Notice

Chapter 11

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 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 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 for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.

Contents