FULL STACK WEB DEVELOPER PART XV

Ruby on Rails

intro to the most famous Web framework

Panos M.

www.techcareerbooster.com

Full Stack Web Developer Part XV: Ruby on Rails

Panos Matsinopoulos

This book is for sale at http://leanpub.com/full-stack-web-developer-part-xv-ruby-on-rails

This version was published on 2019-08-21

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 Tech Career Booster - Panos M.

Contents

The Bundle and the TCB Subscription **1**
Part of Bundle 2
Goes with a TCB Subscription 3
Each TCB Subscription Goes with the Bundle 4

Credits To Photo on Cover Page **5**

About and Copyright Notice **6**

Ruby on Rails **7**
1 - Getting Started With Rails 8
Summary 8
Learning Goals 8
Introduction 9
Main Parts of a Ruby on Rails Web Application 9
Hello Rails! 10
Rails is an MVC Framework 18
HTTP Request Processing 19
Another Application Example - A Blog 28
Integrate Twitter Bootstrap 80
Reminder for Github 83
Deploy to Heroku 83
Closing Note 90
Tasks and Quizzes 90
2 - The Rails Way 92
Summary 92
Learning Goals 92
Introduction 93
PostgreSQL Database on Local Machine 93
Environment Variables 102
Routes with `resources` 104
Form Builders 106
Path and URL Helpers 111
Strong Parameters 114
Model Validations 115
Handle Error 116
Improving Error Handling 117
Informing User 120

Style Div With Error Messages . . . 121
Removing `create.html.erb` Confirmation Page . . . 123
Display Success Message . . . 123
Using Layout For `flash` . . . 125
Anchors With `link_to` . . . 126
Delete Links . . . 127
Delete With Confirmation . . . 128
Using `before` Hooks In Controller . . . 129
Using Partials . . . 132
Rails Console . . . 133
HAML - HTML Abstraction Markup Language . . . 134
Closing Note . . . 140
Tasks and Quizzes . . . 140
3 - Authentication . . . 142
Summary . . . 142
Learning Goals . . . 142
Introduction . . . 143
Basic Authentication . . . 143
Our `blog` Basic Authentication . . . 143
Session Based Authentication . . . 145
Sign Up Form . . . 150
Some Links To Ease Navigation . . . 154
Sign In . . . 154
Current User & Sign Out . . . 159
Feature Consistency . . . 163
Authenticated Pages For Articles . . . 169
Returning Back To Private Page After Sign In . . . 173
Hiding Links That Do Not Apply . . . 176
Partial For Errors . . . 178
Form Similar Code . . . 183
Form Properties Derive From Model State . . . 185
Remove Duplication Once More . . . 186
Closing Note . . . 189
Tasks and Quizzes . . . 189
4 - Sending out Emails . . . 191
Summary . . . 191
Learning Goals . . . 191
Introduction . . . 192
Sending Out Emails While In Development . . . 192
Email is Pending Confirmation . . . 196
`User` Model Changes . . . 199
Display Unconfirmed Flag . . . 201
Sending Email Out . . . 201
EndPoint To Accept Confirmation . . . 203
Amend `SignUp` Controller . . . 204
`User#confirm_email_address` . . . 206

Sending Emails Out - The Recommended Way . . . 207
Ask For Email Address Confirmation Email To Be Resent . . . 211
Indexes In Database . . . 215
SMTP Server For Production Environment . . . 222
Closing Note . . . 223
Tasks and Quizzes . . . 223
5 - Authorization . . . 224
Summary . . . 224
Learning Goals . . . 224
Introduction . . . 225
Binding Articles To Users . . . 225
Schema Migration . . . 226
Rolling Back Latest Schema Migration . . . 228
Foreign Key Name and Index . . . 231
Add Reference Does It All . . . 233
Avoid Repetition in Migration . . . 235
Updating `Article` and `User` Models . . . 237
Updating the Controller . . . 240
Displaying Owner Next To Each Article Entry . . . 241
Editing/Updating an Article . . . 242
Hide Edit Link For Other User's Articles . . . 244
Hide Links In List Of Articles . . . 244
Drying Out Code . . . 248
Other Authorization Gems . . . 251
Closing Note . . . 251
Tasks and Quizzes . . . 251
6 - Testing Rails Applications . . . 252
Summary . . . 252
Learning Goals . . . 252
Introduction . . . 253
Test Folder Tree . . . 253
Test Environment . . . 255
Ruby on Rails Test Framework Class Hierarchy . . . 258
Testing Models . . . 259
Fixtures . . . 266
System Testing . . . 266
Integration Testing . . . 274
Controller Tests . . . 276
Testing Mailers . . . 277
Using `RSpec` . . . 279
Using Cucumber . . . 284
Integration Tests . . . 287
System Requirements . . . 287
Factory Girl . . . 289
Closing Note . . . 293
Tasks and Quizzes . . . 293

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

7 - More About Models . . . 296
Summary . . . 296
Learning Goals . . . 296
Introduction . . . 298
Active Record Quick Overview . . . 298
Schema Migrations . . . 304
Validations . . . 315
Active Record Callbacks . . . 334
Active Record Associations . . . 342
Active Record Query Interface . . . 347
Models that are not Active Records . . . 376
Closing Note . . . 376
Tasks and Quizzes . . . 377
8 - RoR - Ruby Language Extensions . . . 378
Summary . . . 378
Learning Goals . . . 378
Introduction . . . 378
Extensions To All Objects . . . 379
Extensions To `Module` . . . 383
Extensions to `Class` . . . 383
Extensions to String . . . 385
Extensions to Numeric . . . 389
Extensions to Enumerable . . . 390
Extensions to Hash . . . 391
Extensions to Date . . . 393
Extensions to DateTime . . . 394
Closing Note . . . 394
Tasks and Quizzes . . . 395
9 - Internationalizing Your Application . . . 396
Summary . . . 396
Learning Goals . . . 396
Introduction . . . 397
The Most Important Methods . . . 397
Getting Current Locale . . . 398
Getting Available Locales . . . 398
Supporting Another Locale . . . 398
The Default Locale . . . 400
Setting The Locale Per Request . . . 401
Setting The Locale Dynamically . . . 404
Passing Variables To Translations . . . 418
Displaying Date/Time . . . 420
Validation Error Messages . . . 427
User Time Zone . . . 429
Timestamps . . . 432
User Time Zone . . . 432
Closing Note . . . 439

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

Tasks and Quizzes . . . 439
10 - Background Jobs . . . 441
Summary . . . 441
Learning Goals . . . 441
Introduction . . . 442
Introducing Background Runner . . . 443
Some Properties Of Background Runner . . . 444
ActiveJob . . . 445
Define a Job . . . 445
Enqueue The Job . . . 448
Other Backends . . . 448
Delayed Job Backend . . . 448
Queues . . . 456
Callbacks . . . 459
Mailers . . . 463
Job Testing . . . 464
Scheduling Jobs To Run At Specific Points in The Future . . . 464
Background Runners in Heroku . . . 464
Closing Note . . . 465
Tasks and Quizzes . . . 465
Task Details . . . 465
11 - JavaScript with Rails Applications . . . 466
Summary . . . 466
Learning Goals . . . 466
Introduction . . . 467
Introduction to Ajax . . . 467
jQuery and Ajax . . . 468
Rails and Ajax . . . 477
Action Cable . . . 479
Closing on ActionCable . . . 499
Closing the Rails Section . . . 499
Tasks and Quizzes . . . 500

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

The Bundle and the TCB Subscription

Part of Bundle

This book is not sold alone. It is part of the bundle Full Stack Web Developer.

Goes with a TCB Subscription

When you purchase the bundle, then you have full access to the contents of the TCB Courses.

Each TCB Subscription Goes with the Bundle

Moreover, this goes vice-versa. If you purchase the subscription to the TCB Courses, then you are automatically eligible for the Full Stack Web Developer bundle.

Credits To Photo on Cover Page

Image by Dimitris Vetsikas from Pixabay

About and Copyright Notice

Full Stack Web Developer - Part XV - Ruby on Rails 1st Edition, August 2019

by Panos M. for Tech Career Booster (https://www.techcareerbooster.com)

Copyright (c) 2019 - Tech Career Booster and Panos M.

All rights reserved. This book may not reproduced in any form, in whole or in part, without written permission from the authors, except in brief quotations in articles or reviews.

Limit of Liability and Disclaimer of Warranty: The author and Tech Career Booster have used their best efforts in preparing this book, and the information provided herein "as is". The information provided is delivered without warranty, either express or implied. Neither the author nor Tech Career Booster will be held liable for any damages to be caused either directly or indirectly by the contents of the book.

Trademarks: Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

For more information: https://www.techcareerbooster.com

Ruby on Rails

Ruby on Rails is one of the most famous Web development frameworks out there. It has also been used as a leverage to boost the popularity of Ruby programming language. It is full of tools and techniques that will both make your life much easier as a Web developer and, at the same time, teach you how to do Web development the proper way. Lots of other Web frameworks are now trying to follow Ruby on Rails. But, RoR is going to be the leader in many years to come. Learning RoR will ease your learning paths for other Web frameworks too.

1 - Getting Started With Rails

Summary

Getting Started With Rails

This is going to be a long chapter on Ruby on Rails. It is still, however, an introduction to this fantastic Web framework. The chapter will take you through all the basic components of a Rails application. You will build two applications while studying the content and a third one while doing the chapter tasks. This last one will be evolving with the knowledge that you will be acquiring going through the next chapters. The content here is so powerful that will make you feel really awesome with the things you will learn and the actual results of the hands-on material practiced.

Learning Goals

1. Learn the basics of Ruby on Rails.
2. Learn about being DRY.
3. Learn about convention over configuration.
4. Learn how to install Ruby on Rails.
5. Learn how to start and stop Rails server.
6. Learn to read the Rails server logs.
7. Learn how to display your own welcome page.
8. Learn that Rails is an MVC framework.
9. Learn how an HTTP request is being processed from component to next.
10. Learn about routes.
11. Learn how to print the table of routes.
12. Learn how a route entry points to a controller and an action.
13. Learn where and how you define the controllers and its actions.
14. Learn how a view corresponds to a controller and an action.
15. Learn about the layouts and how views are embedded inside them.
16. Learn about resources.

17. Learn about CRUD and HTTP APIs.
18. Learn how to create a resource.
19. Learn what does it take submit a form and make sure you follow the best security practices.
20. Learn how to initialize the database your Web application is bound to.
21. Learn how to generate a schema migration.
22. Learn how to write a migration to define the columns a table should have.
23. Learn how to define the models of your Web application.
24. Learn how to build a show and a list page for a resource.
25. Learn how you use instance variables from controllers to display dynamic information in the views.
26. Learn how you build a page to update the details of an existing resource.
27. Learn how you can use HTTP verbs that browsers do not really support.
28. Learn how to implement the feature of deletion of a resource.
29. Learn how to integrate Twitter Bootstrap to your Rails application.
30. Learn how to deploy your application to Heroku.

Introduction

Ruby on Rails is a Web framework that will help you develop a Web application quickly and using the latest Web technologies. It is written in Ruby, of course, and for that reason it requires that you understand and practice Ruby.

Rails is an opinionated software. This means that it has taken some decisions before you do and believes that if you follows things Rails suggests, then you will be hugely productive and happy. This also means that some of the practices that you may used to from other languages or frameworks might not be applicable in Rails world.

Rails favours DRY (Do not Repeat Yourself) and Convention over Configuration. DRY is important because when you keep things without code duplication, the maintenance effort is hugely decreased. On the other hand, if you have the same code repeated again and again, a small change might be carried over in multiple places in your code. With regards to Convention over Configuration, Rails believes that we should not rely on huge configuration files (see other frameworks that use XML configuration) that would tell us how Web application should be configured, but we should rely on conventions. Developing using conventions means that you get things for free without having to change or set up anything. In this chapter and in the chapters that follow, it will become very obvious to you, how conventions in Rails help you do things very quickly with minimum typing.

> **Important**: In this chapter we will not build the perfect Rails application. Many things that we will show you can be done much easier with Rails. But we want you to learn what happens behind the scenes so that you can understand the magic that comes with Rails.

Main Parts of a Ruby on Rails Web Application

Before we start our first project with Rails, let's see what are the main parts of Rails Web Applications.

Rails server

The Rails server is the executable that will bring an HTTP Server on, at a specific port ready to accept and serve connections from a client, usually a browser. We saw a similar thing with Rack and Sinatra in the *Introduction to Web Development* course.

We will see later on how we start Rails Server

RDBMS

Rails is designed to work with an RDBMS (Relational Database Management System). It integrates with many RDBMS products, out of the box. The following is a list of known RDBMS systems that integrate with Rails out of the box:

- MySQL
- Oracle
- PostgreSQL
- SQlite3
- FrontBase
- IBM DB
- MS SQL Server
- and many databases that support JDBC.

In order to build a Web application you will need to decide on your RDBMS product[*]. For the purposes of this course we will use SQLite3 and PostgreSQL.

> *Your SQL Knowledge:* If you don't know the basics of SQL you will have problem developing Web applications with Ruby on Rails. You can learn the basics if you take the section *Relational Database Management Systems - SQL*.

[*] Rails integrates with NoSQL database products too, like MongoDB and Neo4J. These will be presented in next sections of this course.

Hello Rails!

Let's create the classic *Hello World* project. In fact, it will be *Hello Rails*. This will quickly set everything for you to create bare minimum Ruby on Rails application.

Set `rvm`

We will not start a RubyMine project yet. We will first set the `rvm` gemset for our new project. Create a gemset with name `hello_rails` and Ruby `2.X`. This is how I did that:

```
$ rvm use 2.3.3@hello_rails --create
ruby-2.3.3 - #gemset created /Users/panayotismatsinopoulos/.rvm/gems/ruby-2.3.3@h\
ello_rails
ruby-2.3.3 - #generating hello_rails wrappers..........
Using /Users/.../.rvm/gems/ruby-2.3.3 with gemset hello_rails
$
```

(the above code snippet online)

Then, double check that your current `rvm` is the one just created:

```
$ rvm current
ruby-2.3.3@hello_rails
$
```

(the above code snippet online)

Install `rails`

Rails will be installed as a gem. It will be installed in your gemset with the following command:

```
$ gem install rails --no-ri --no-rdoc
Fetching: i18n-0.8.1.gem (100%)
Successfully installed i18n-0.8.1
Fetching: thread_safe-0.3.6.gem (100%)
Successfully installed thread_safe-0.3.6
Fetching: tzinfo-1.2.3.gem (100%)
Successfully installed tzinfo-1.2.3
Fetching: concurrent-ruby-1.0.5.gem (100%)
Successfully installed concurrent-ruby-1.0.5
Fetching: activesupport-5.1.1.gem (100%)
...
Successfully installed sprockets-rails-3.2.0
Fetching: rails-5.1.1.gem (100%)
Successfully installed rails-5.1.1
36 gems installed
$
```

(the above code snippet online)

When you run the above command, `gem install rails --no-ri --no-rdoc`, you will see a huge list of gems that are installed alongside with `rails`. Rails gem depends on many other gems. One of the dependencies is `bundler` gem, which is also installed.

Having done the installation as above, let's confirm that we have the `rails` executable ready to be invoked:

```
$ rails --version
Rails 5.1.1
$
```

(the above code snippet online)

Great! At the time of this writing, Rails was on version 5.1.1. The above command confirms that we can invoke the `rails` executable.

Ask Rails to Create Your Application

Now that we have `rails` installed, lets ask Rails to create the folder of our application. This is how we should do that:

```
$ rails new hello_rails
      create
      create  README.md
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
         run  git init from "."
Initialized empty Git repository in /Users/.../hello_rails/.git/
      create  app
      create  app/assets/config/manifest.js
      create  app/assets/javascripts/application.js
...
      create  package.json
      remove  config/initializers/cors.rb
      remove  config/initializers/new_framework_defaults_5_1.rb
         run  bundle install
...
Fetching gem metadata from https://rubygems.org/.............
Fetching version metadata from https://rubygems.org/...
Fetching dependency metadata from https://rubygems.org/..
Resolving dependencies...
Installing rake 12.0.0
Using concurrent-ruby 1.0.5
...
Using rails 5.1.1
Installing sass-rails 5.0.6
Bundle complete! 16 Gemfile dependencies, 70 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
         run  bundle exec spring binstub --all
* bin/rake: spring inserted
* bin/rails: spring inserted
$
```

(the above code snippet online)

The command `rails new hello_rails` did a lot of things.

1. Firstly, it created the folder `hello_rails` and a big number of files and folders inside it.
2. One of the files that it has created inside the `hello_rails` folder was the `Gemfile`. It put inside it all the gems that your application will be relying on, besides the `rails` gem.
3. Moreover, with the `Gemfile` in place, it ran `bundle install` and installed all the gems inside your gemset.
4. Also, it initialized your application to be `git` ready, by having executed the command `git init`. Moreover it created some files that are `git` related. The `.gitignore` files tell `git` which files will be ignored.
5. Finally, it altered the `bin/rake` and `bin/rails` scripts to be integrated with `spring`. `spring` is a files pre-loader that is very useful while doing test driven development, because it pre-loads your application files and your tests run faster.

Switch To Your Application Folder

With `rails` having done the hard work to set up your application folder, you can now change directory to this new folder that has been created.

```
$ cd hello_rails
hello_rails $
```

(the above code snippet online)

Freeze `rvm` Settings

Before we continue let's make sure that we have frozen the `rvm` settings, by creating the two files that integrate our project with `rvm`.

```
hello_rails $ echo '2.3.3' > .ruby-version
hello_rails $ echo 'hello_rails' > .ruby-gemset
hello_rails $ cd .
hello_rails $ rvm current
ruby-2.3.3@hello_rails
```

(the above code snippet online)

Do your first Commit

Also, let's integrate with `git`:

> **Important**: Before you add and commit, make sure that you update the `.gitignore` file to not include the `.idea` folder that is used by RubyMine.

```
hello_rails $ git add .
hello_rails $ git commit -m "Initial Commit after rails app initialization"
[master (root-commit) 85c3dac] Initial commit after Rails app initialization
 78 files changed, 1194 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 .ruby-gemset
...
 create mode 100644 test/test_helper.rb
 create mode 100644 tmp/.keep
 create mode 100644 vendor/.keep
hello_rails $
```

(the above code snippet online)

Integrate with Github

Then create a new repository in your Github account and make sure that you add the Github remote. This is how I have added the Github remote to my project:

```
hello_rails $ git remote add origin git@github.com:pmatsinopoulos/hello_rails.git
hello_rails $ git push -u origin master
Counting objects: 86, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (70/70), done.
Writing objects: 100% (86/86), 20.74 KiB | 0 bytes/s, done.
Total 86 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To github.com:pmatsinopoulos/hello_rails.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
hello_rails $
```

(the above code snippet online)

Now my project has a git remote. This can be confirmed with:

```
hello_rails $ git remote -v
origin  git@github.com:pmatsinopoulos/hello_rails.git (fetch)
origin  git@github.com:pmatsinopoulos/hello_rails.git (push)
hello_rails $
```

(the above code snippet online)

Start and Navigate To Your Application

We are ready to start our Rails server:

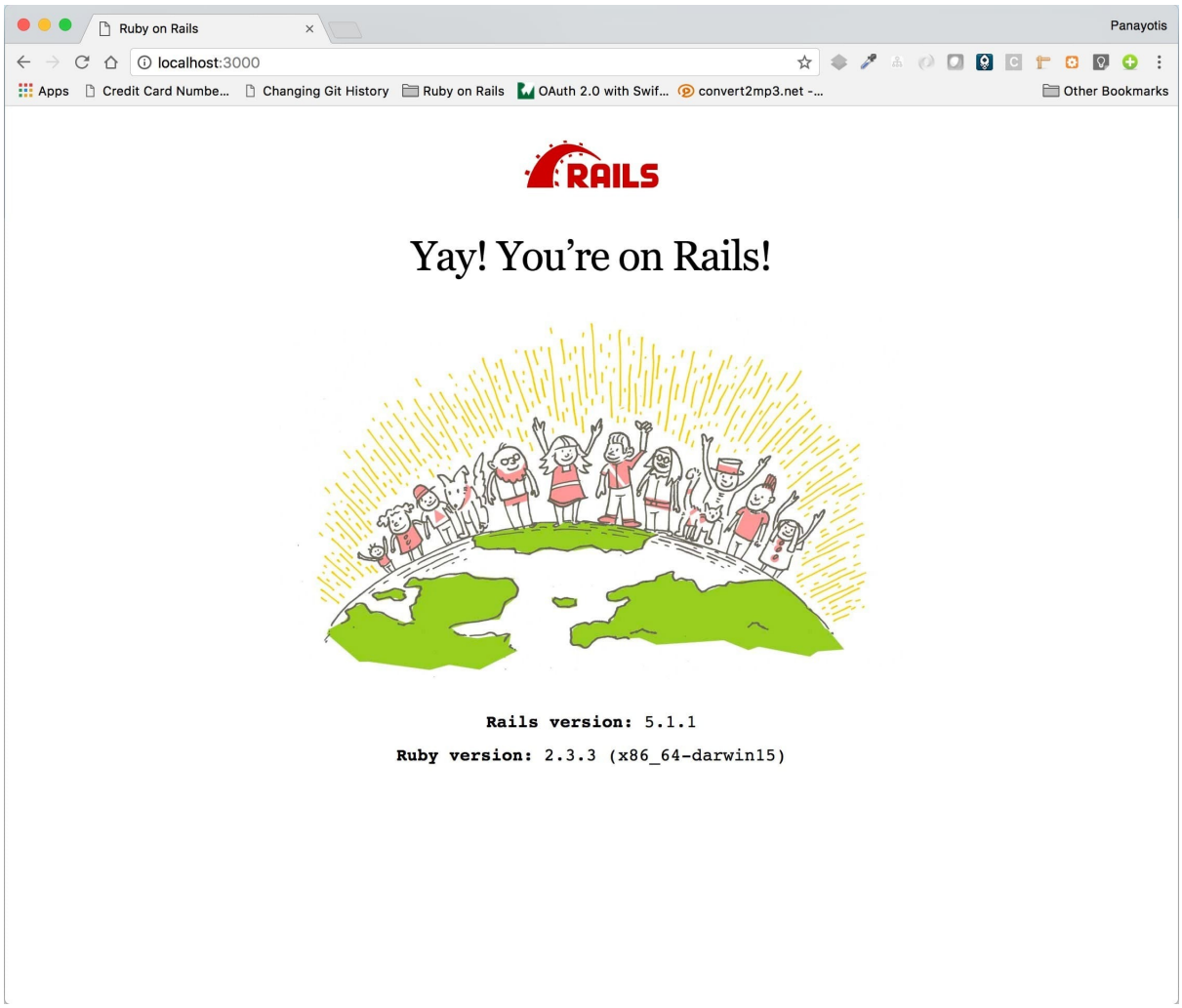
```
hello_rails $ bin/rails server
=> Booting Puma
=> Rails 5.1.1 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

(the above code snippet online)

You can see that your Rails server is now listening on port `3000`. Also, it tells you that you can stop your Rails server if you click on <kbd>Ctrl+C</kbd>.

Also, you can see that the Web server that is hosting our Web application is `Puma` and the environment is `development`. We will talk more about environments later on, but if you took the *Sinatra* chapter, environments work more or less in a similar manner.

With the Rails server up and running, let's visit the page http://localhost:3000. You will see this:



Default Rails Welcome Page

Bingo! You have just sent a request to Ruby on Rails server, to your Web application server.

If you see the messages printed at the Rails terminal, you will see these lines added:

```
Started GET "/" for 127.0.0.1 at 2017-05-14 08:45:18 +0300
Processing by Rails::WelcomeController#index as HTML
  Rendering .../.rvm/gems/ruby-2.3.3@hello_rails/gems/railties-5.1.1/lib/rails/te\
mplates/rails/welcome/index.html.erb
  Rendered .../.rvm/gems/ruby-2.3.3@hello_rails/gems/railties-5.1.1/lib/rails/tem\
plates/rails/welcome/index.html.erb (5.3ms)
Completed 200 OK in 220ms (Views: 11.3ms)
```

(the above code snippet online)

You can tell from the first line that the `GET "/"` has been served and some other information that we will talk you about in the next chapters.

Change the Welcome Page

Rails uses a default welcome page for newly created applications. How can we change this?

We will let you know here what actions you will need to take to change this. Then we will expand in explaining how this one works.

1. Open the file `config/routes.rb` and make sure the content is as follows:

```
1. # File: config/routes.rb
2. #
3. Rails.application.routes.draw do
4.   root 'welcome#index'
5. end
```

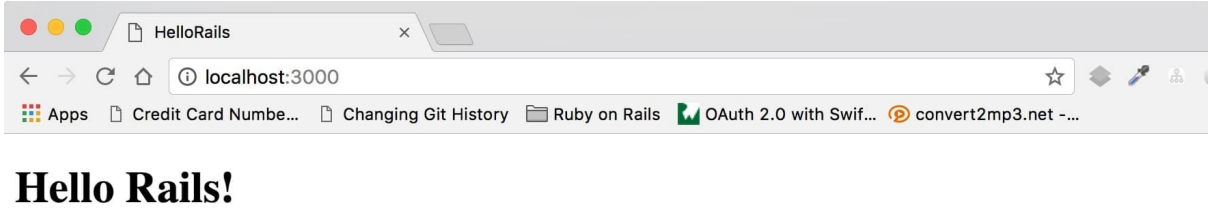
2. Create the file `app/controllers/welcome_controller.rb`. Watch out, it is the file `welcome_controller.rb` inside the folder `controllers` which is a folder inside the `app` folder.

```
1. # File: app/controllers/welcome_controller.rb
2. #
3. class WelcomeController < ApplicationController
4. end
```

3. Create the file `app/views/welcome` and the file `app/views/welcome/index.html.erb` with the following content:

```
<% # File: app/views/index.html.erb %>
<h1>Hello Rails!</h1>
```

Now you are ready to go. Visit the http://localhost:3000 page again.

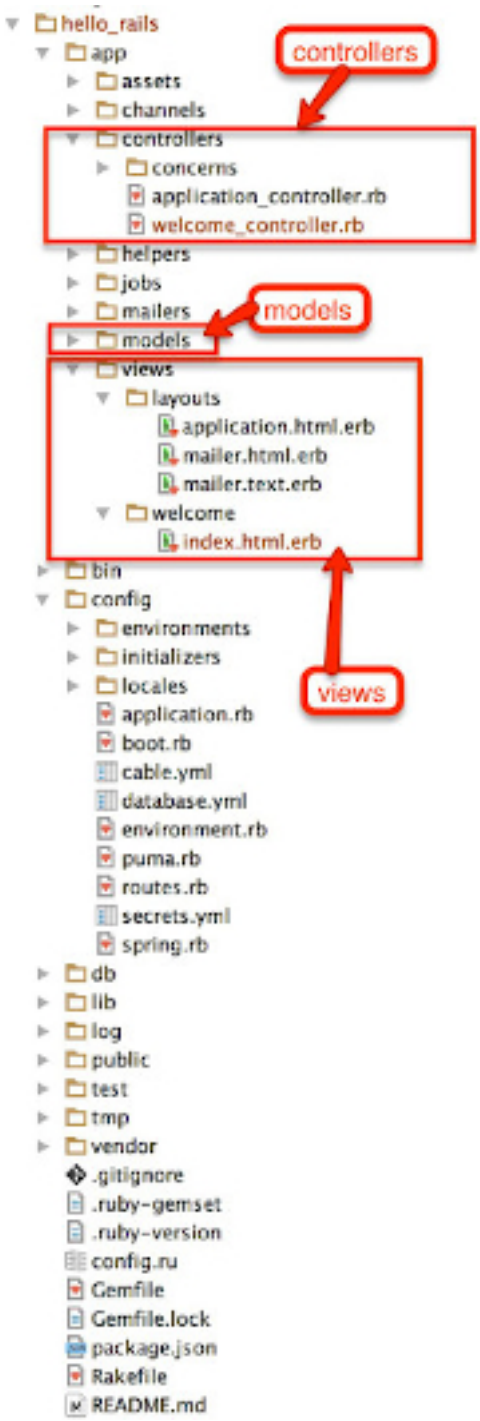


Hello Rails Welcome Page

Rails is an MVC Framework

Rails is an MVC (Model View Controller) Web framework.

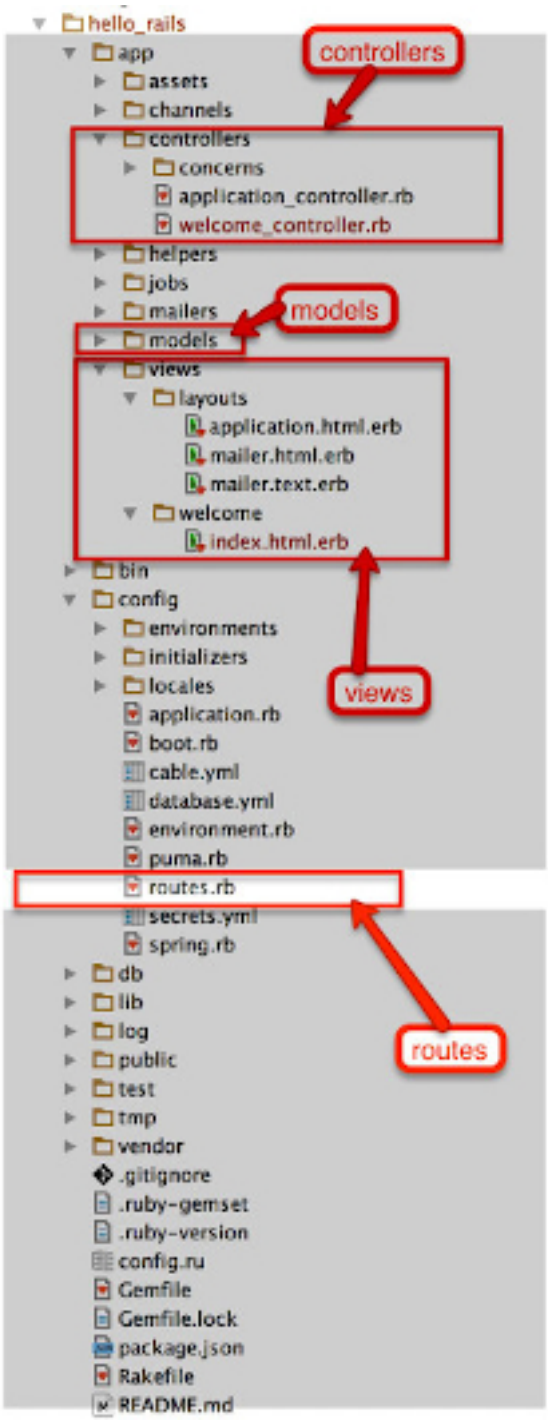
1. The Views reside inside the folder tree `app/views`. The view is an HTML-like file. It is a file with extension `.html.erb`. This means that it is a template file that uses the ERB (Embedded Ruby) language.
2. The controllers reside inside the folder tree `app/controllers`.
3. The models reside inside the folder tree `app/models`.



Models, Views, Controllers and Corresponding Folder Trees

HTTP Request Processing

Each HTTP request is being processed by a group of pieces of code that include at least controllers and views. Usually, processing includes models too. However, there is also a very important piece of code too that is called *routing*. It is whatever we put inside the `config/routes.rb` file:

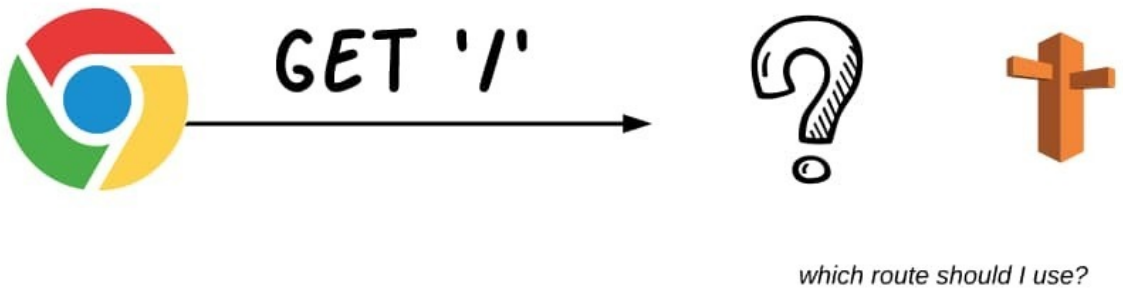


Routes Is Another Important Component

Let's see how the `GET /` for the `http://localhost:3000` *hello_rails* application has been processed.

Match a Route Entry

Firstly, Rails will try to match the incoming HTTP verb + path combination to a route entry. Since, when we load the page `http://localhost:3000/`, the path is `/` and the HTTP verb is `GET`, we need to match `GET '/'` to a routes entry.



Which Route Should I Use?

Currently our routes are defined as such:

```
# File: config/routes.rb
#
Rails.application.routes.draw do
  root 'welcome#index'
end
```

(the above code snippet online)

There is only one entry. Does it match to `GET '/'`? Yes it does. This is what the `root` method call corresponds to.

And if you want to get the actual routes that your application supports, and the ones that are actually and magically created by the `config/routes.rb` file above, you can do this: Open a terminal at the root folder of your project and type the following command:

```
hello_rails $ bin/rake routes
Running via Spring preloader in process 4449
Prefix Verb URI Pattern Controller#Action
  root GET  /           welcome#index
hello_rails $
```

(the above code snippet online)

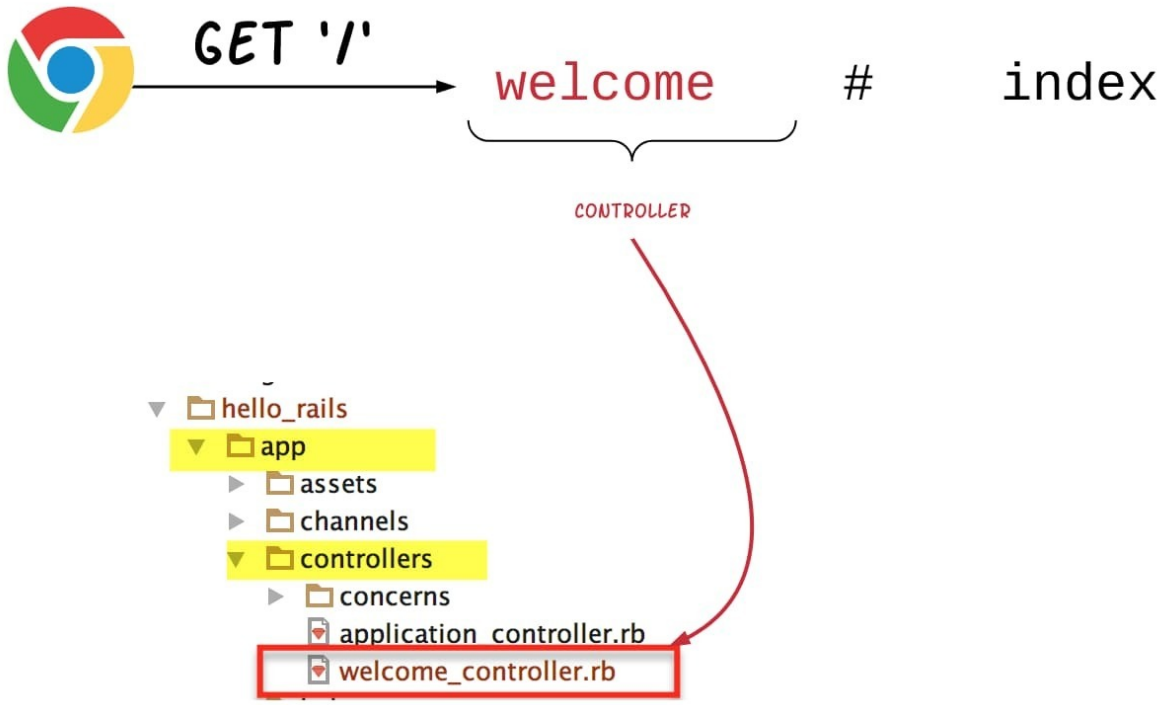
You can see that the `GET` verb alongside the `/` URI Pattern, it has only one entry in the routes table.

Route Entry Matches to A Controller

The routes need to tell Rails which controller will handle the incoming request.

A Route Entry Needs To Tell Rails Which Controller

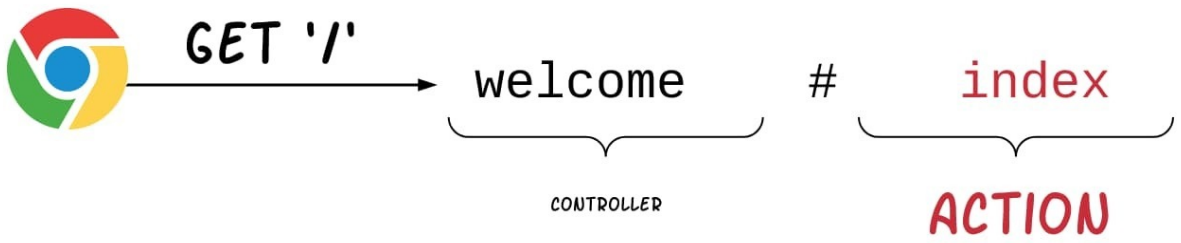
The entry `root GET / welcome#index` tells that the `GET '/'` should be handled by the `welcome` controller. The welcome controller should exist and should be implemented as a class `WelcomeController`. It should exist inside the `app/controllers` folder within the file `app/controllers/welcome_controller.rb`.



Welcome Controller Should Be Located Inside app/controllers Folder

Route Entry Matches to An Action Too

Identifying the controller from the route entry is not enough. Rails needs to identify the action too.

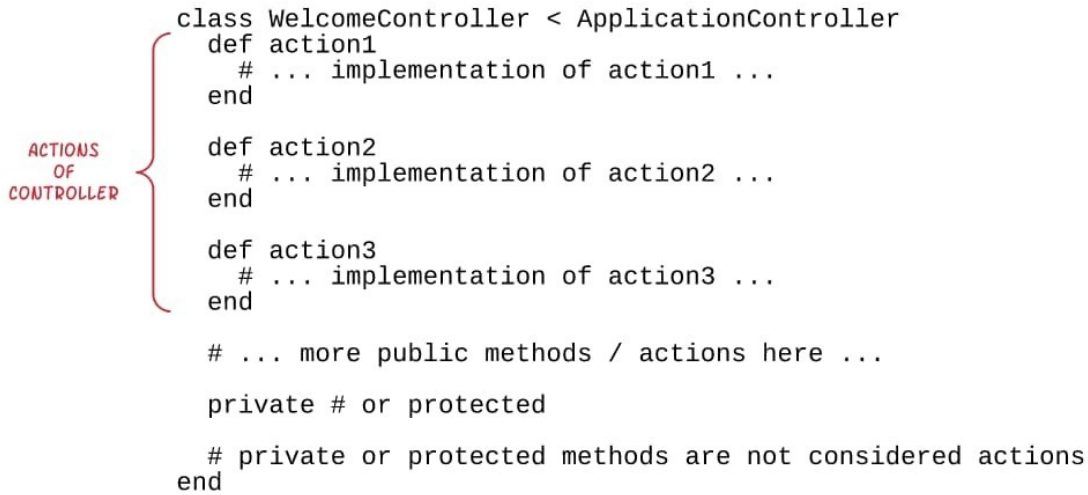


Route Needs To Tell Which Action Of The Controller

The action `index` is specified by the entry `root GET / welcome#index`. The action is specified in the scope of the controller. In other words, the `welcome#index` tells that the request should be handled by the action `index` of the `welcome` controller.

Actions are The Public Methods of Controller

The actions are the public methods of controller.



Actions Are The Public Methods of the Controller

Since our route specifies that the request should be handled by the `index` action, then `WelcomeController` class needs to have a public method that has the name `index`.

Although this is what usually happens, the action is optional. If missing the controller handles the request using a default behaviour. Nevertheless, all the controllers derive from the `ApplicationController` class (file `app/controllers/application_controller.rb`) which, in turn, derives from the `ActionController::Base`. In other words, `WelcomeController` bears a lot of default behaviour inherited from the Rails framework classes, one of which is `ActionController::Base`.

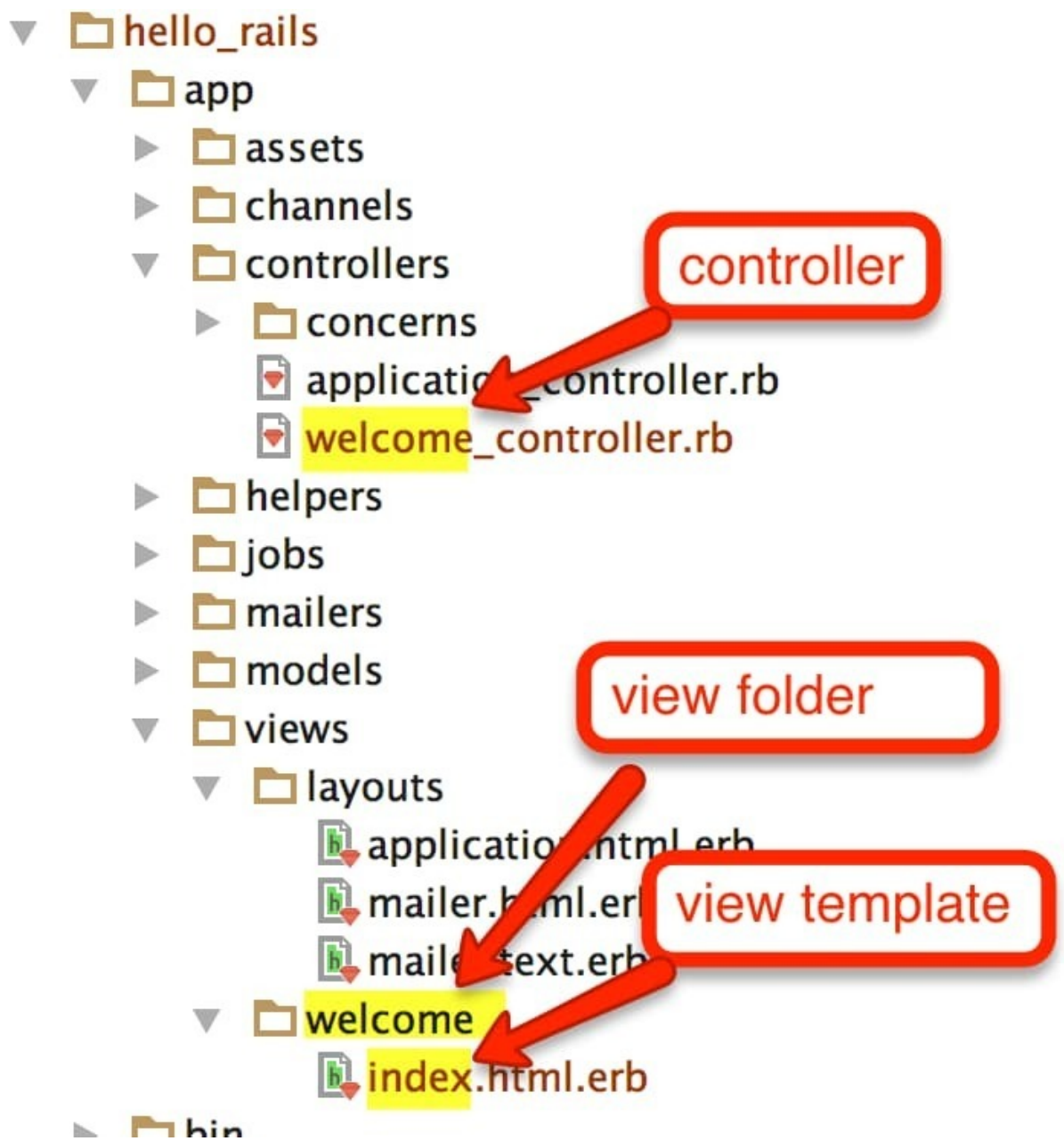
If the action is missing, the controller goes directly to render the corresponding view. Otherwise, it executes the Ruby statements inside the action implementation and then renders the corresponding view.

After Controller, It is the View

After the execution of the Ruby statements inside the action (public method) of the Controller, Rails renders the corresponding view. What is the corresponding view for the Controller with name `welcome` and action `index`?

- Rails looks inside the `app/views` sub-folders for a folder with same name as the controller name, i.e. with name `welcome`.
- Then, inside the found folder, it tries to locate the file with same name as the action name, i.e. with name `index`. The file name extension should be something that allows Rails to render an HTML content. By default, `.html.erb`. This means it is a file that is a view template, that uses the template language ERB (Embedded Ruby). The ERB template language allows us to generate dynamic content within the view.

  ERB and Dynamic Content: This idea has been explained thoroughly in the Sinatra chapter, section *Introduction to Web Development*.

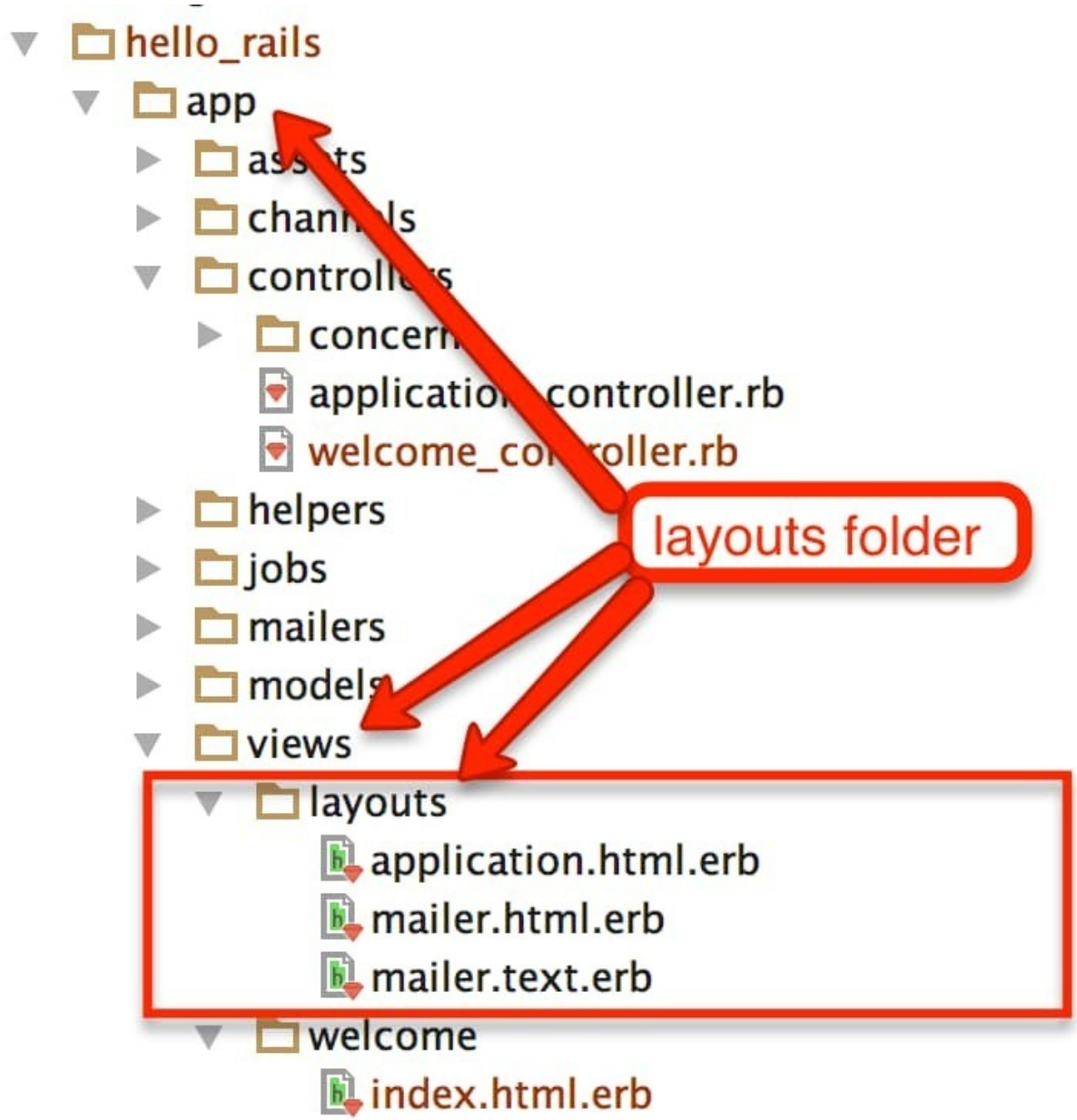


Controller, View Folder and View Template

View Embedded In Layout

If one looks carefully the content of the `app/views/welcome/index.html.erb`, they will see that this is not a proper HTML page. It is only an HTML snippet. How does Rails view renderer know how to return back to the requesting client a proper HTML page?

It uses the *layouts*. The layout is a special view template file that resides inside the `app/views/layouts` folder.



Layouts Folder

You can have more than one layout. However, the default behavior Rails has in order to pick up the layout is

1. Tries to find the layout file with same name as the controller name. Hence, it tries to find inside the `app/views/layouts` folder the file with name `welcome.html.erb`.
2. If this does not exist, it tries to find a layout with the name equal to the parent of the `welcome` controller. Since the parent class of `WelcomeController` is the class `ApplicationController`, it tries to locate the file `app/views/layouts/application.html.erb`.

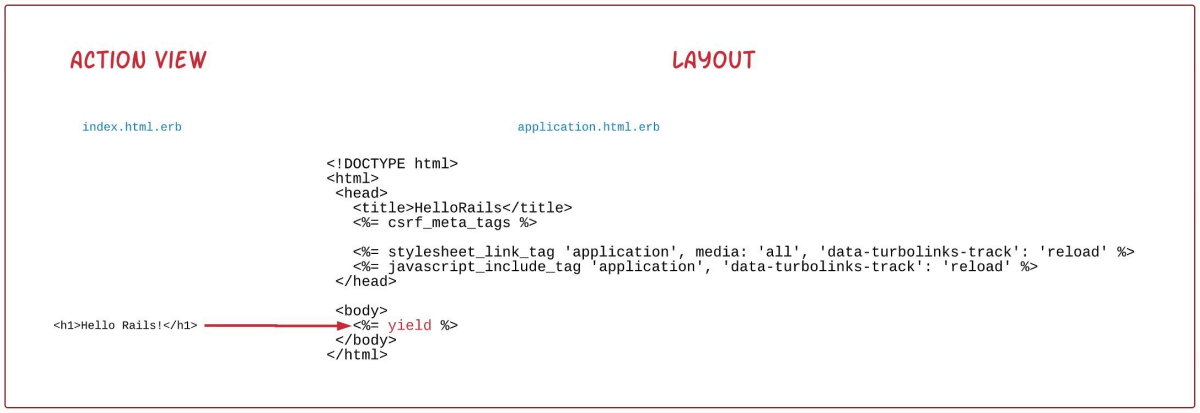
In our example, the `app/views/layouts/application.html.erb` file exists and has the following content:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>HelloRails</title>
5      <%= csrf_meta_tags %>
6
7      <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-trac\
8  k': 'reload' %>
9      <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
10   </head>
11
12   <body>
13     <%= yield %>
14   </body>
15 </html>
```

(the above code snippet online)

This looks more like a proper HTML file. It does have embedded Ruby code that dynamically contributes to the final HTML content, but, nevertheless, if you study that, you will identify the parts that make this a proper HTML document.

Besides the other parts of the layout that you can see in the above file, the one that is related to embedding the view inside the layout, is the line 12, with the `<%= yield %>` command. This is the point of the layout that will be replaced with the corresponding view template code.



Action View File Content is Embedded in Layout

Convention Over Configuration

That was an overview of the HTTP request processing by Rails framework and your Web application that sits on top of it. Also, the whole explanation above might have rung a bell to you about what Rails considers *convention* (over configuration).

You just experienced convention as follows:

1. Controllers live inside the `app/views/controllers` folder. E.g. `app/views/controllers/welcome_controller.rb`.

2. Actions of controllers that handle incoming requests are public methods.
3. View files that correspond to actions of controllers live in the `app/views` folder tree, and in particular inside a sub-folder that has same name as the name of the controller. E.g. `app/views/welcome`.
4. The view files themselves, they have the same name as the action that they correspond to. E.g. `app/views/welcome/index.html.erb`.
5. Views are embedded inside a layout. The layout lives inside the folder `app/views/layouts`. It has the same name as the controller, e.g. `app/views/layouts/welcome.html.erb` or a name equal to the name of a parent class of the controller that handles the incoming request, e.g. `app/views/layouts/application.html.erb`.

Rails is full of conventions and the above were only some examples of this idea. Learning and following these conventions will make you a much happier developer.

Double Check with Logs

While you are sending requests from your browser to your Rails server, server prints useful information on the terminal that it has been started on. Also, it writes those logs inside the `log` folder. Look for the file `development.log`, which is the file in which logs are written while we are in *development* environment.

For example, look what it is printed when you go to the page http://localhost:3000:

```
Started GET "/" for 127.0.0.1 at 2017-05-14 17:59:52 +0300
Processing by WelcomeController#index as HTML
  Rendering welcome/index.html.erb within layouts/application
  Rendered welcome/index.html.erb within layouts/application (0.3ms)
Completed 200 OK in 19ms (Views: 16.3ms)
```

(the above code snippet online)

1. It tells you that the `GET "/"` is handled by the controller `WelcomeController` and its action `index`.
2. It tells you that it renders the view template `welcome/index.html.erb` within the layout `layouts/application.html.erb`.
3. It tells you that it returns the HTTP response status code `200 OK`.
4. It also gives you some numbers with regards to how long it spent processing the request. In the example above, it was 19ms out of which 16.3ms was spent in rendering the view.

Get into the habit to read the logs frequently. Logs will give you insight on how Rails works and how it processes requests. You will be able to troubleshoot and understand the behaviour of your application.

Another Application Example - A Blog

We will continue our initial familiarization with Rails by developing a Blog application.

`rvm` gemset

Firstly, we will initialize `rvm` for the `blog` Rails application. Move to the root folder in which you usually create your projects. Then create the necessary `rvm` gemset. This is what I did:

```
$ rvm use 2.3.3@blog --create
ruby-2.3.3 - #gemset created /Users/.../.rvm/gems/ruby-2.3.3@blog
ruby-2.3.3 - #generating blog wrappers..........
Using /Users/.../.rvm/gems/ruby-2.3.3 with gemset blog
$ rvm current
ruby-2.3.3@blog
```

(the above code snippet online)

Install `rails`

Being in the correct gemset, we install `rails`.

```
$ gem install rails --no-ri --no-rdoc
...
Successfully installed rails-5.1.1
36 gems installed
$
```

(the above code snippet online)

Initialize `blog` with `rails`

Now that we have Rails installed, let's initialize our `blog` application.

```
$ rails new blog
      create
      create  README.md
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
         run  git init from "."
...
Using sass-rails 5.0.6
Bundle complete! 16 Gemfile dependencies, 70 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
         run  bundle exec spring binstub --all
* bin/rake: spring inserted
* bin/rails: spring inserted
$
```

(the above code snippet online)

This will create the folder `blog` with all the necessary folders and files in order for you to start your blog application development in Rails.

Freeze `rvm` Settings

Now we `cd` to our new application folder, `blog`, and we create the files necessary for `rvm` to freeze Ruby version and gemset.

```
$ cd blog
blog $ echo '2.3.3' > .ruby-version
blog $ echo 'blog' > .ruby-gemset
blog $ rvm current
ruby-2.3.3@blog
blog $
```

(the above code snippet online)

Integration With `Git`

> **Important**: Before you add and commit files, make sure that the file `.gitignore` has one entry for `.idea` so that your `git` command line ignores the folder with the files that are generated and managed by RubyMine.

You can now do `git add .` and then `git commit ...` to create your first commit with all the files of your application in.

```
blog $ git add .
blog $ git commit -m "Initial Commit"
[master (root-commit) 53c7d76] Initial Commit
 78 files changed, 1194 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 .ruby-gemset
 create mode 100644 .ruby-version
...
 create mode 100644 test/system/.keep
 create mode 100644 test/test_helper.rb
 create mode 100644 tmp/.keep
 create mode 100644 vendor/.keep
blog $
```

(the above code snippet online)

Integrate with Github

Create a new Github repository and add a new remote to your local git configuration. This is how I did that:

```
blog $ git remote add origin git@github.com:pmatsinopoulos/blog.git
```

(the above code snippet online)

Then push your first commit to remote:

```
blog $ git push -u origin master
```

(the above code snippet online)

Resources

Rails is heavily based around *resources*. A *resource* is a set of similar things. For example in our blog application a resource is the *Article*. A *resource* is something that is being managed by the Web application. When we say *managed* we mean *created*, *retrieved*, *updated* and *deleted* (CRUD).

> **RESTful Services**: Since we have just talked about resources, you may want to have a look at *REST* and what it is really about. One of the properties of REST APIs is that they are *resource-based*. We have found this reference here being quite accurate and worth watching.

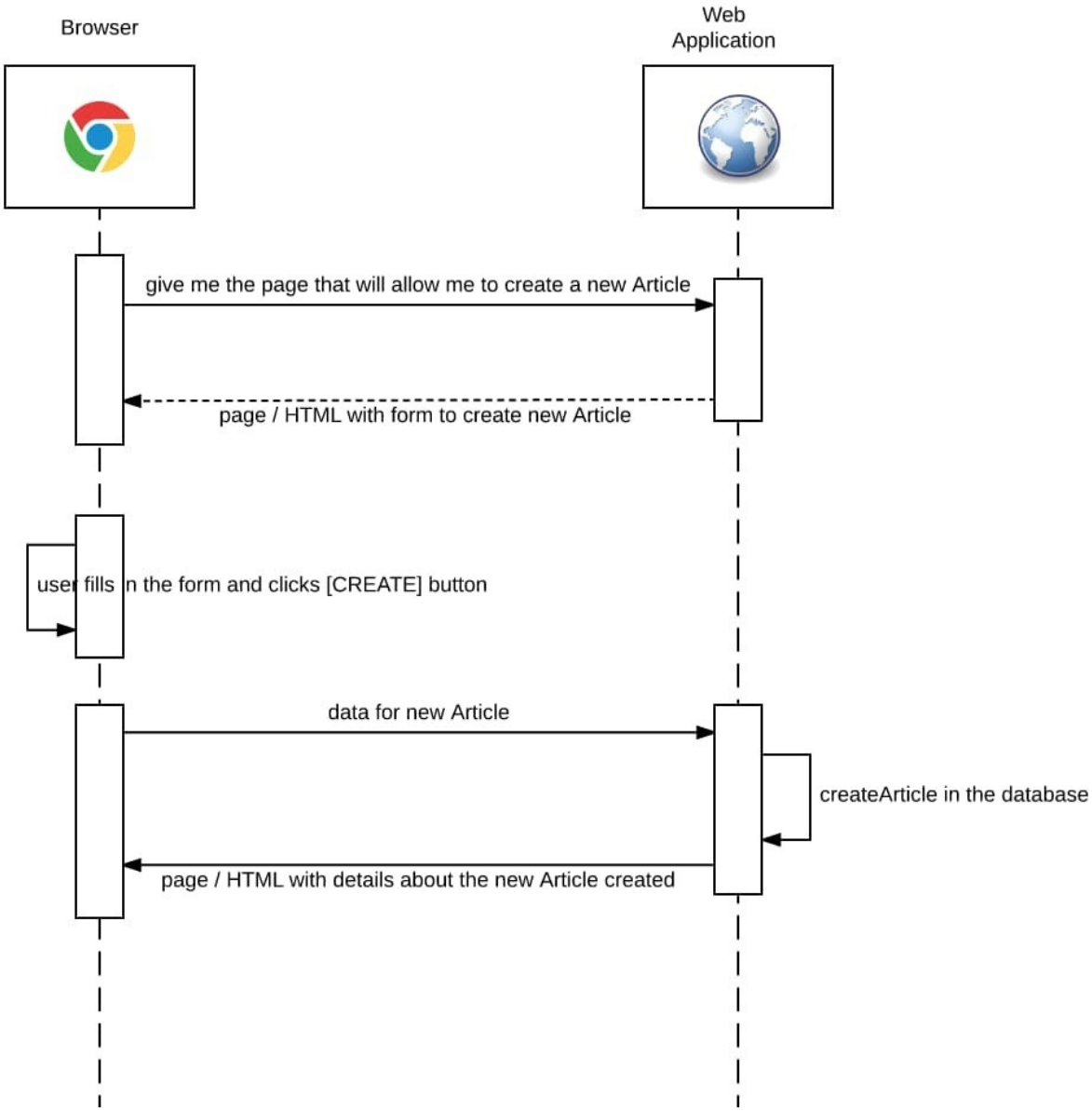
CRUD and HTTP API

Our Web application is a server application that has to talk HTTP to the clients connected. If we want to follow the principles of RESTful architectures, we need our Web application to expose an HTTP API that is resource-based. Let's see what we really mean with this, by assuming that our blog application needs to *manage* an *Article* resource.

Create - (C)

User should be able to create an Article. Usually, when a user wants to create an Article, first, they need to be presented with a page that would allow them to enter and submit the details of an Article. In fact, the interaction of the user browser to the Web application is more or less this:

1. Browser asks Web application to return back the page that will allow the user to enter and submit the Article details.
2. Browser submits the Article details to the endpoint that the page specifies.
3. Web application takes the details of the Article and creates the new Article.
4. Web application returns back to the browser the content of a new page that informs the user that the Article has been created.



Browser Interaction To Create an Article

Getting Page With Article Creation Form

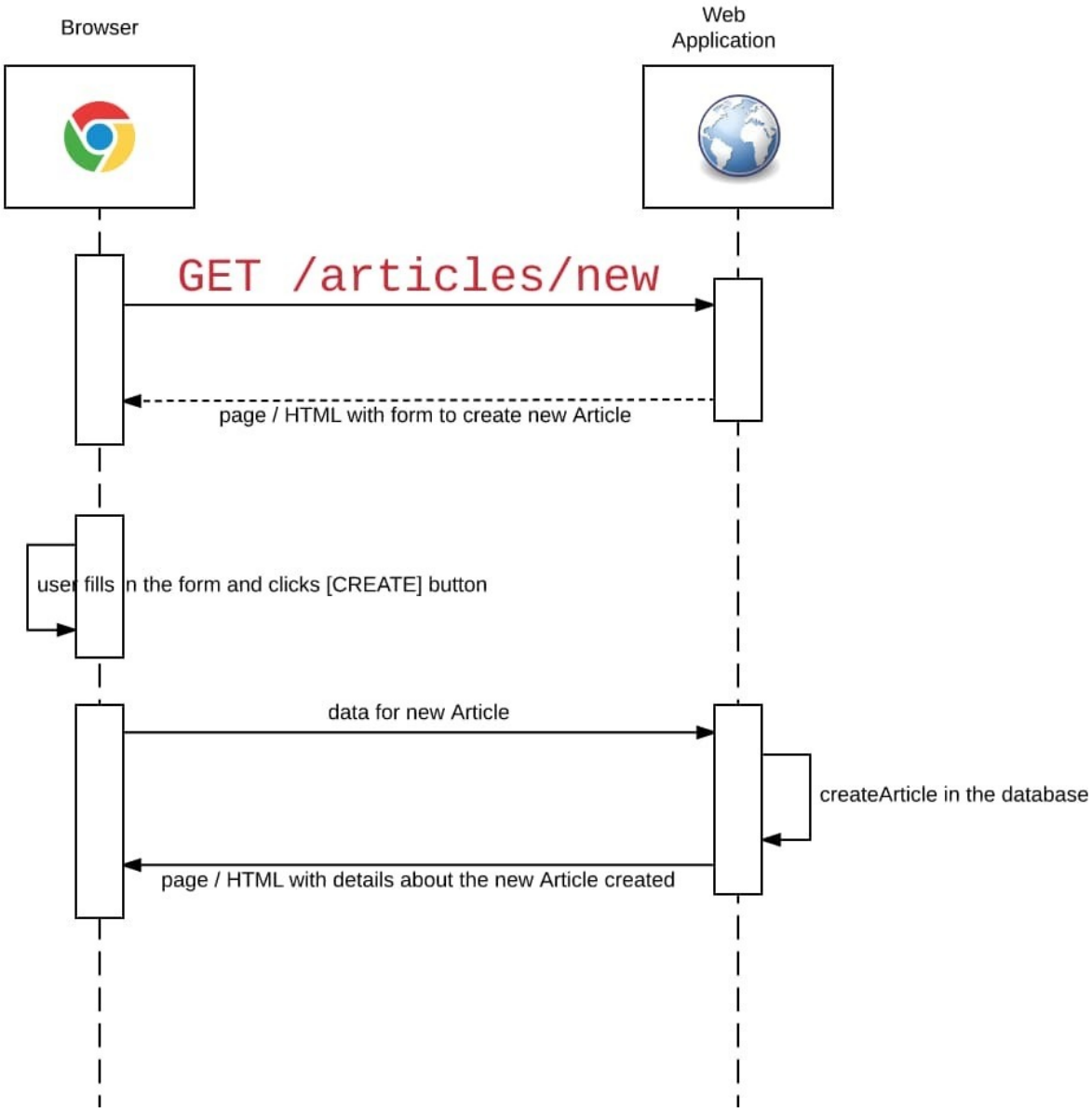
As you can see above, the first thing a browser does is to ask Web application to return back the page that will allow the user to enter and submit details of the new Article. How is this request, `give me the page that will allow me to create a new Article`, translated to technical terms?

It is an HTTP request. Correct? And all HTTP requests need to include an HTTP Verb and a URI that will identify the resource the verb will act upon.

What about this?

```
GET /articles/new
```

We use the verb `GET` because we want to retrieve a page. We do not want to change the state of the resource. We use the `/articles/new` to tell our Web application that we want to create a new Article. As simple as that!



GET /articles/new

How is this translated to Rails terms? We need to create the correct *route* inside the `config/routes.rb` file. This is how we declare the endpoints and the controllers + actions to handle the HTTP requests.

Let's do it. This is the new version of the `config/routes.rb` file in your `blog` project:

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  get 'articles/new'
end
```

(the above code snippet online)

I guess that the line `get 'articles/new'` is quite self-explanatory now.

Let's run the command that prints the current routes of our Web application so that we understand which endpoints are being served by which controllers and actions:

```
blog $ bin/rake routes
Running via Spring preloader in process 12870
      Prefix Verb URI Pattern             Controller#Action
articles_new GET  /articles/new(.:format) articles#new
blog $
```

(the above code snippet online)

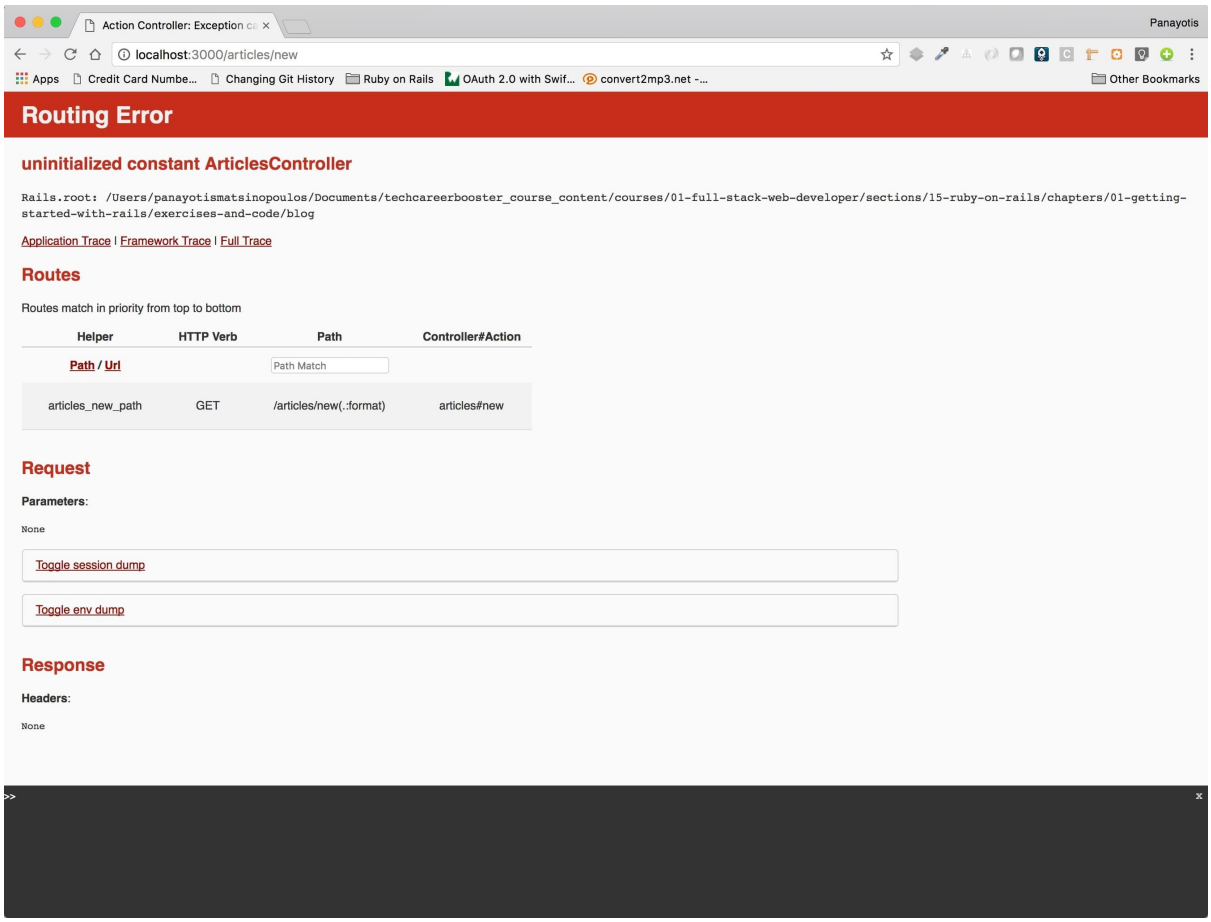
Rails was clever enough to tell from the `config/routes.rb` entry that we have added, which is going to be the controller and its action to handle this request.

If you start your Rails server now ....

```
blog $ bin/rails server
=> Booting Puma
=> Rails 5.1.1 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

(the above code snippet online)

and then visit the page http://localhost:3000/articles/new, you will certainly get an error page like this:



Error Page - Missing ArticlesController

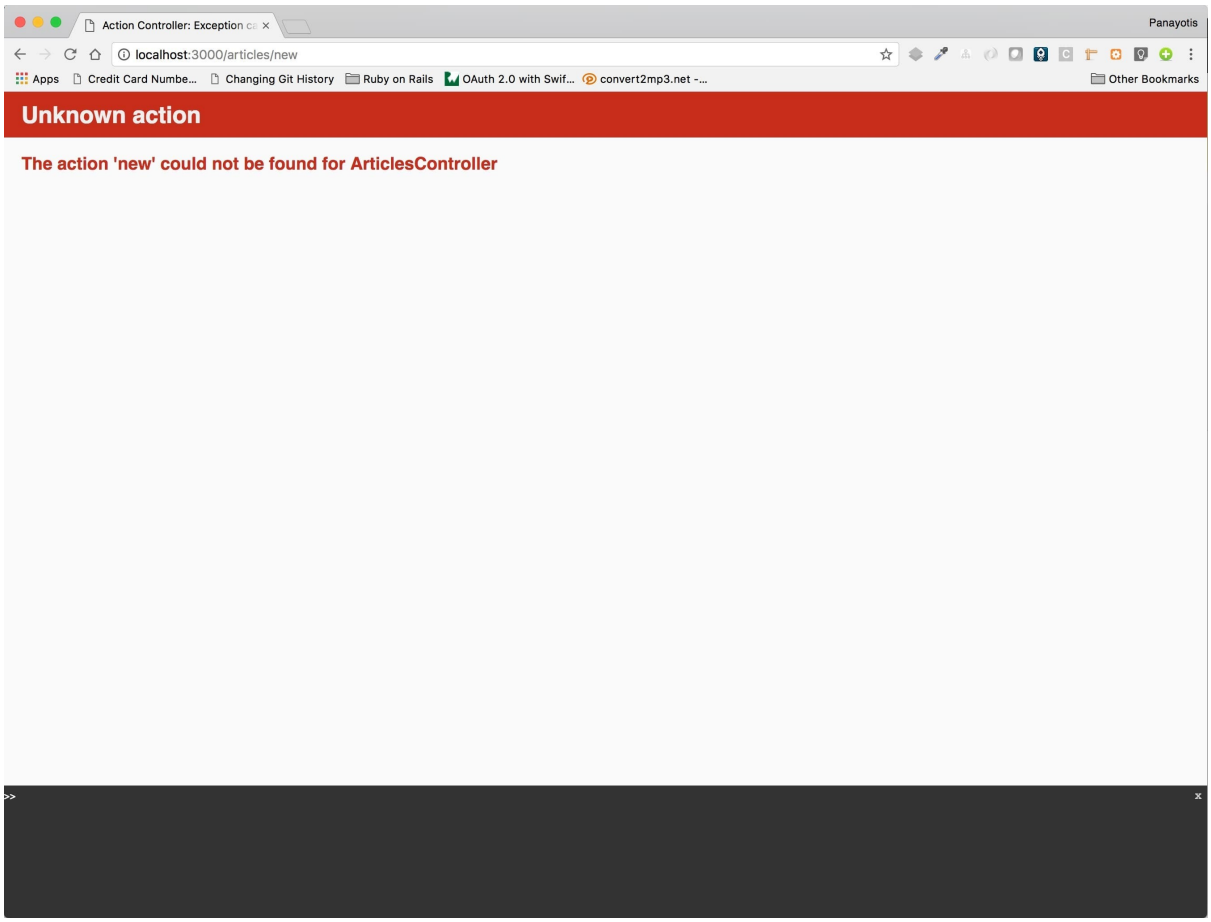
That was expected, because we have not defined our controller, the `ArticlesController` class.

Let's do that. Create the file `app/controllers/articles_controller.rb` in your `blog` application, with the following content:

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
end
```

(the above code snippet online)

When you do that, then reload the page on your browser. You will get this error:



New Action Not Found

That was expected too. We don't have any `new` action defined in the `ArticlesController`.

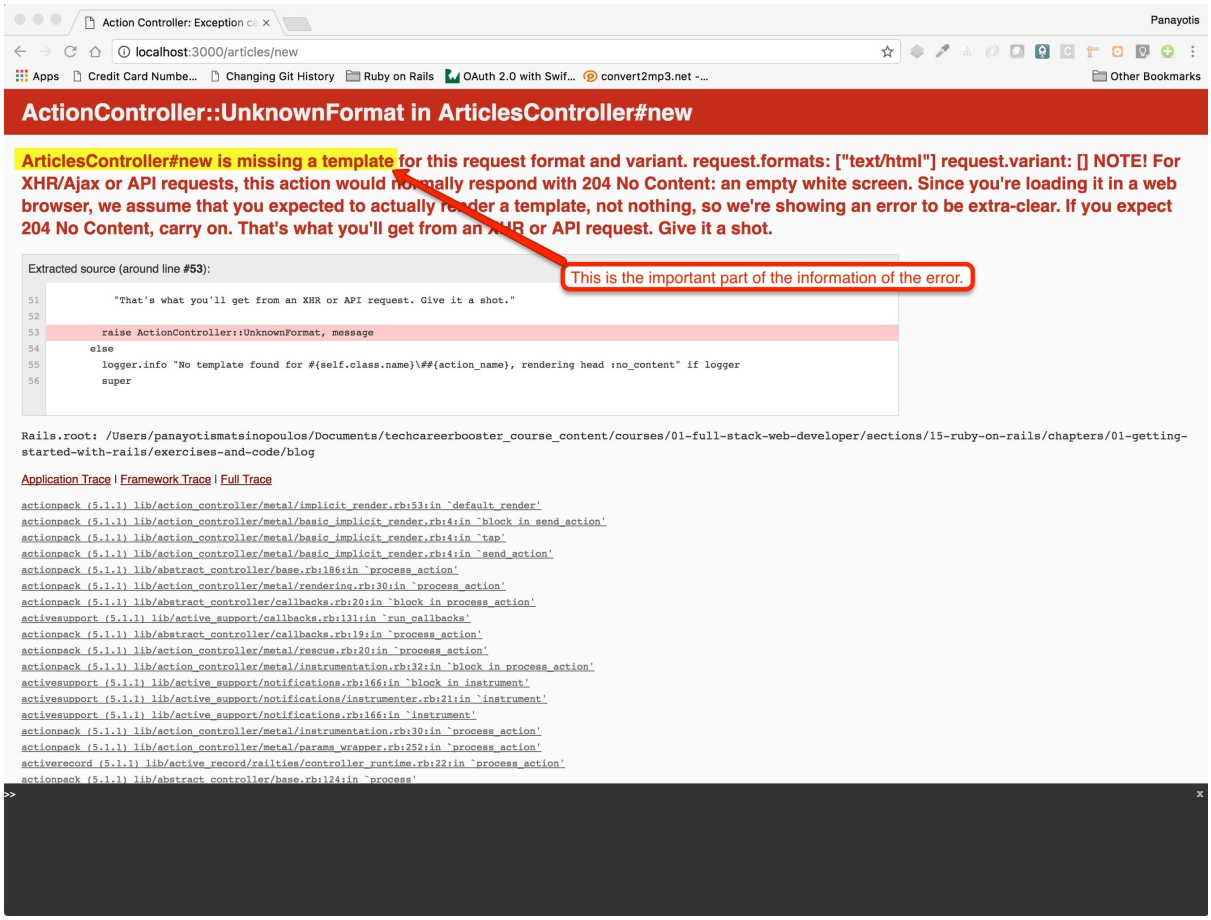
Let's create the action. The action is going to be a public method in the `ArticlesController`:

```
1  # File: app/controllers/articles_controller.rb
2  #
3  class ArticlesController < ApplicationController
4    def new
5    end
6  end
```

(the above code snippet online)

Lines 4 and 5, above, define the `new` action inside the `ArticlesController`. We currently have this as an empty action.

Let's reload the page on our browser. This is what we will see:



New Action is Missing a Template

That was expected too. Remember that we need a view template in order to build an HTTP response with HTML content.

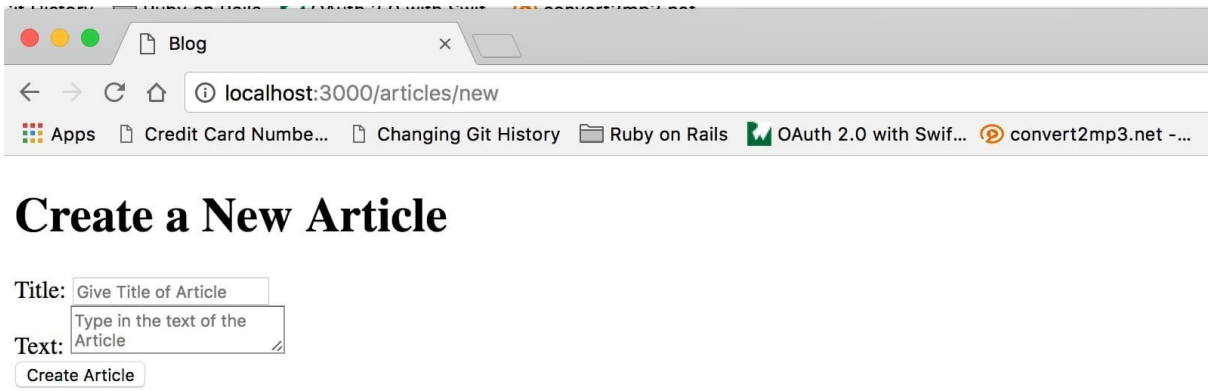
Let's create the view template now. Create the folder `app/views/articles` and then the file `app/views/articles/new.html.erb` with the following content:

```
<% # File: app/views/articles/new.html.erb %>

<h1>Create a New Article</h1>
<form>
  <div>
    <label for="title">Title:</label>
    <input type="text" id="title" name="title" placeholder="Give Title of Article\
"/>
  </div>

  <div>
    <label for="text">Text:</label>
    <textarea id="text" name="text" placeholder="Type in the text of the Article"\
></textarea>
  </div>
```

```
16
17    <div>
18      <button type="submit">Create Article</button>
19    </div>
20  </form>
```

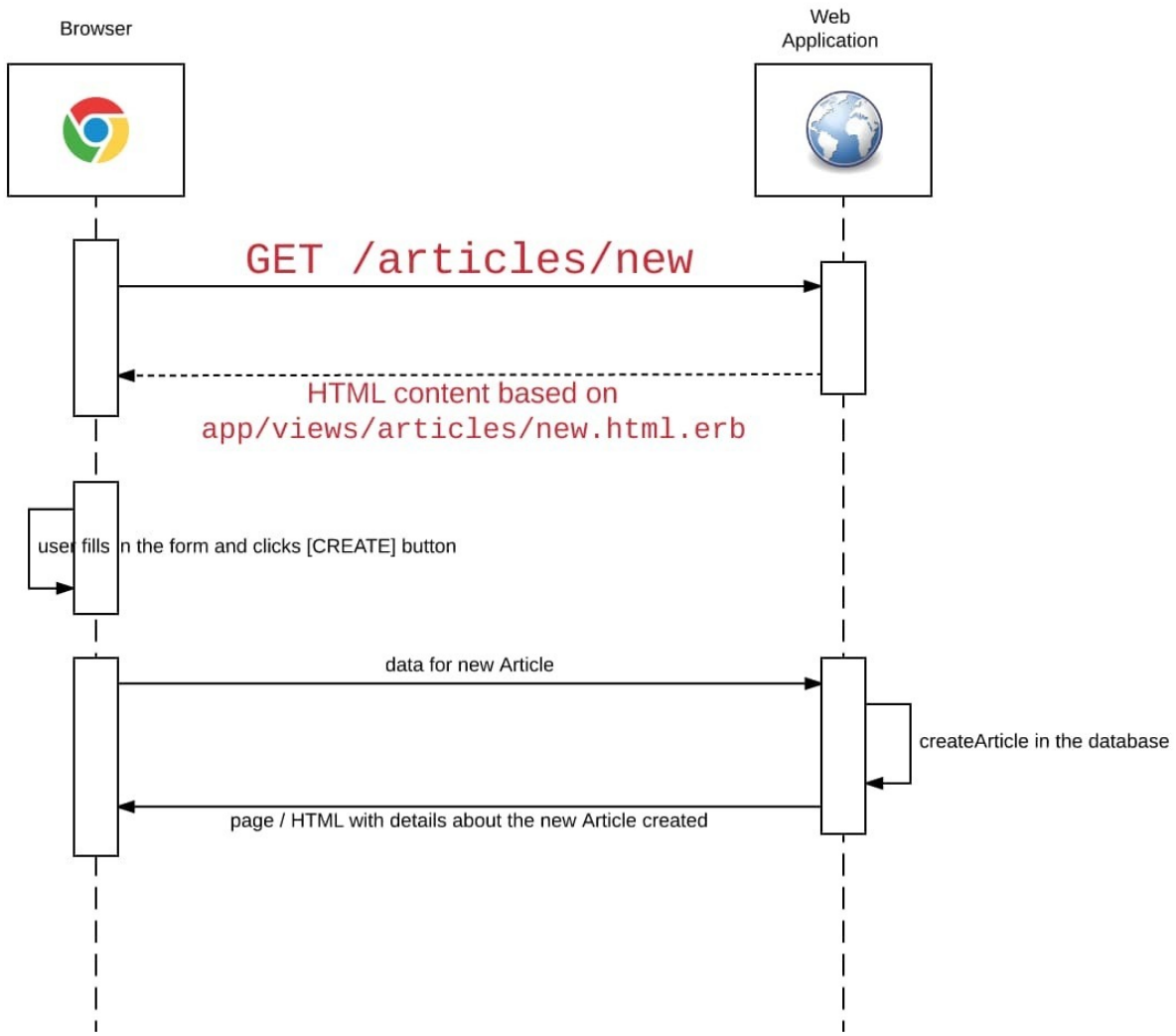
(the above code snippet online)

This is pretty straightforward. Save the above and reload the page on your browser. You will see something like this:



Page With Form To Create A New Article

And this is the point at which we have implemented the first version of the Web application response to the `GET /articles/new` request:



Web App Returns HTML Content To Allow For Creation of new Article

Submitting the Form

But, the form is missing stuff. It cannot be used for submission like this. Two things that are necessary are the `method` and the `action`. The `method` needs to be the HTTP verb that will tell Web application that data submitted should be used for the *creation* of a new Article. `action` needs to be the URI that would tell us which resource the HTTP request refers to.

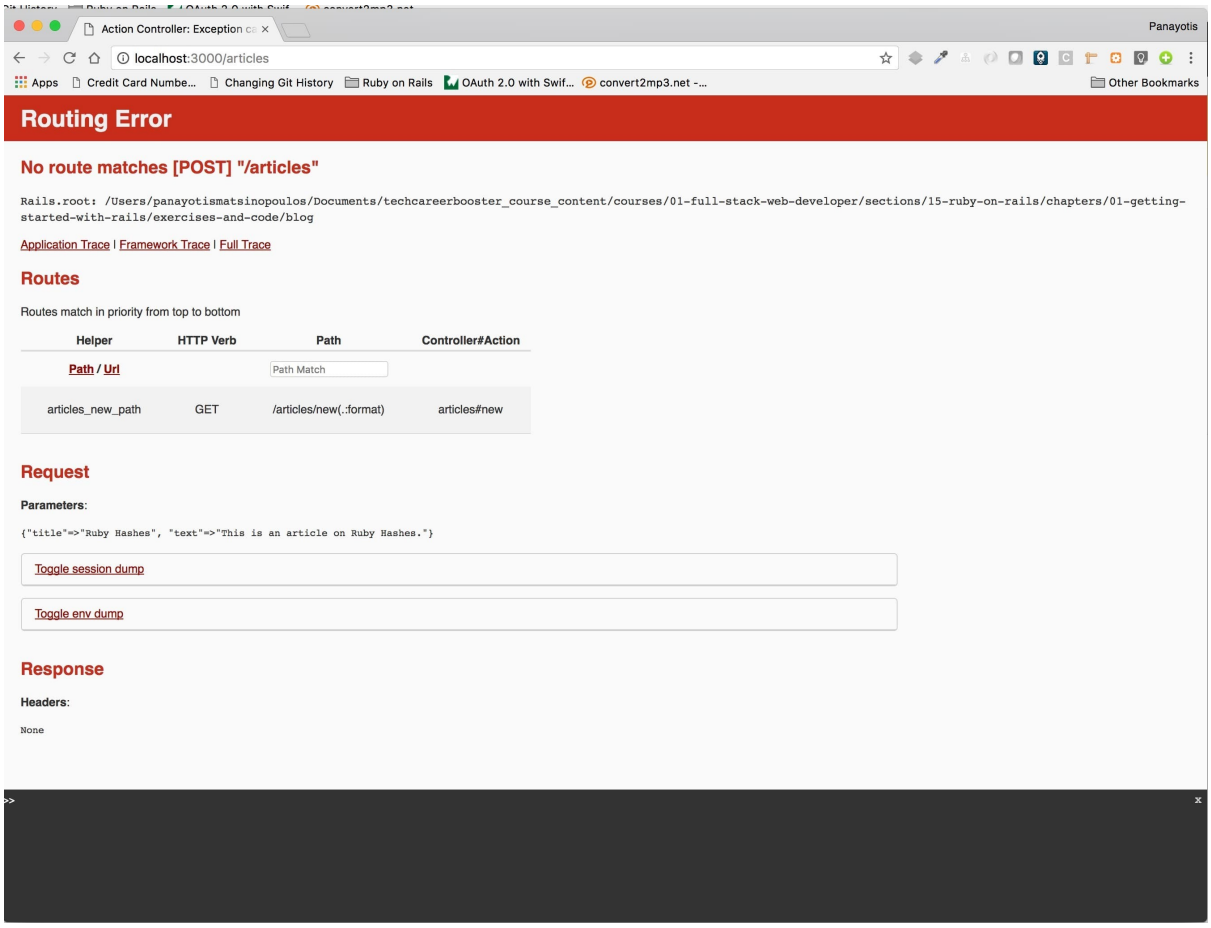
Let's enhance our form to be as follows:

```
1  <% # File: app/views/articles/new.html.erb %>
2
3  <h1>Create a New Article</h1>
4  <form method="post" action="/articles">
5    <div>
6      <label for="title">Title:</label>
7      <input type="text" id="title" name="title" placeholder="Give Title of Article\
8  "/>
9    </div>
10
11   <div>
12     <label for="text">Text:</label>
13     <textarea id="text" name="text" placeholder="Type in the text of the Article"\
14 ></textarea>
15   </div>
16
17   <div>
18     <button type="submit">Create Article</button>
19   </div>
20 </form>
```

(the above code snippet online)

Currently, the only change is on line 4, where we specify the `method` being `post` and the `action` being `/articles`. We have learned that the HTTP verb for creating resources is `POST` and that is why we have set `method` to have the value `post`. The `/articles` is a good resource URI that, with the HTTP verb combined, can clearly tell Web server that we want to create a new Article. In other words, `POST /articles` is the endpoint at which we are going to post the data of the form.

With the above file saved, fill in the form with some data, for example `Ruby Hashes` for `title` and `This is an article for Ruby Hashes` in the `text`, and click on the button `Create Article`. This is what you will get:



Post to Articles Fails With Routing Error

Expected! Our routes does not include any entry to handle the `POST /articles` HTTP request. Let's create one:

```
1  # File: config/routes.rb
2  #
3  Rails.application.routes.draw do
4    get 'articles/new'
5    post 'articles', to: 'articles#create'
6  end
```

(the above code snippet online)

The new entry here is the line 5, `post 'articles', to: 'articles#create'`. Note that although line 4 was enough for Rails to derive the controller and the action, line 5 `post 'articles'` was not enough. That's why we had to add the `to: 'articles#create'` argument too.

Now, if you run the tool to get the list of currently defined routes, you will get this:

```
blog $ bin/rake routes
Running via Spring preloader in process 21312
      Prefix Verb URI Pattern             Controller#Action
articles_new GET  /articles/new(.:format) articles#new
    articles POST /articles(.:format)     articles#create
blog $
```

(the above code snippet online)

Perfect! It seems that we now have the endpoint `POST /articles` covered. It will be handled by `articles` controller `create` action.

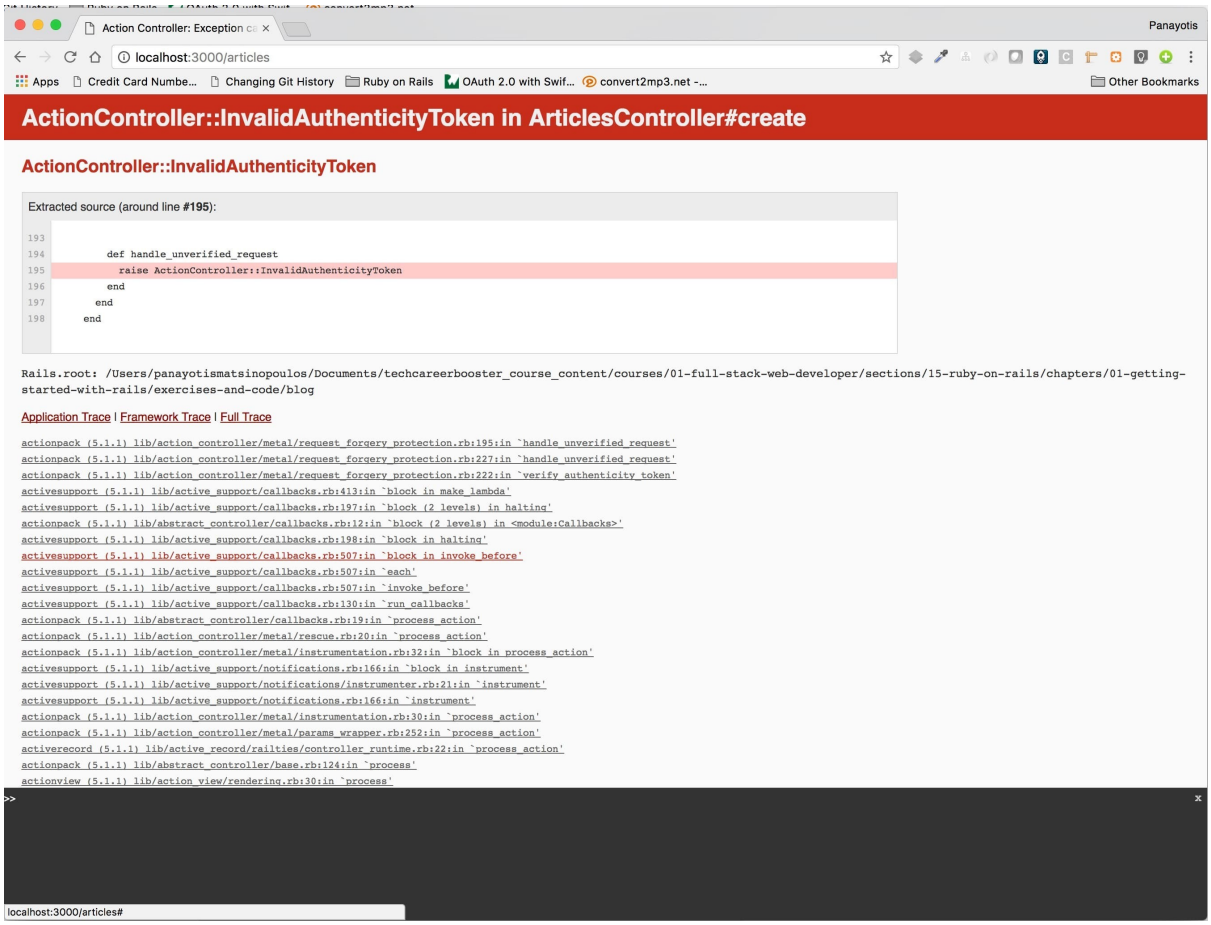
Of course, resubmitting the form, will raise another error, that of the `create` action missing. Let's add this too.

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  def new
  end

  def create
  end
end
```

(the above code snippet online)

Submitting the form will raise an error again. Is it that the `create` action is missing a corresponding view template file. The truth is that this is not the case yet.

Let's try to submit the form. Note that the view template is still missing.



Invalid Authenticity Token

Hmmm! `Invalid Authenticity Token`. This is raised by the Rails framework. When Rails receives a POST request, it makes sure that this is coming from your site, i.e. from the site that actually loaded the form, by checking for the *authenticity token* submitted alongside your data. The authenticity token needs to be a hidden field in your form and needs to have the same value like the one that is carefully hidden inside the *session* that Rails has started with your browser.

But how can we fill in this hidden field with the correct value?

```
1  <% # File: app/views/articles/new.html.erb %>
2
3  <h1>Create a New Article</h1>
4  <form method="post" action="/articles">
5    <input type="hidden" id="authenticity-token" name="authenticity_token" value="..."/>
6    <div >
7      <label for="title">Title:</label>
8      <input type="text" id="title" name="title" placeholder="Give Title of Article"/>
9    </div>
10
11   <div>
12     <label for="text">Text:</label>
13     <textarea id="text" name="text" placeholder="Type in the text of the Ar
14   </div>
15
16   <div>
17     <button type="submit">Create Article</button>
18   </div>
19 </form>
```

we need to fill this with an authenticity value

Authenticity Token As Hidden Field - What Value

There is a method that you can call that will generate the correct authenticity token value. It is the `form_authenticity_token` method. Let's change our view that generates the form to have

this hidden field with the correct value in, generated by this method call:

```
1  <% # File: app/views/articles/new.html.erb %>
2
3  <h1>Create a New Article</h1>
4  <form method="post" action="/articles">
5    <input type="hidden" id="authenticity-token" name="authenticity_token" value="<\
6  %= form_authenticity_token %>"/>
7    <div>
8      <label for="title">Title:</label>
9      <input type="text" id="title" name="title" placeholder="Give Title of Article\
10 "/>
11   </div>
12
13   <div>
14     <label for="text">Text:</label>
15     <textarea id="text" name="text" placeholder="Type in the text of the Article"\
16 ></textarea>
17   </div>
18
19   <div>
20     <button type="submit">Create Article</button>
21   </div>
22 </form>
```

(the above code snippet online)

Do you see line 5? We have amended it so that the value for the `authenticity_token` is generated at the server-side using the method `form_authenticity_token`.

Now, if you submit the form, you will not actually see the error, but, yet, you will not see anything changing. You will see again the same form rendered with the values that you have keyed in intact. So, what did actually happen?

If you look carefully at the Rails server logs, you will see something like this:

```
1 Started POST "/articles" for 127.0.0.1 at 2017-05-18 22:50:58 +0300
2 Processing by ArticlesController#create as HTML
3   Parameters: {"authenticity_token"=>"jSW02x6mO4jXUDOgzey5BY0GCISC9YyMYFX6mJhpsAO\
4 1EPuOZLKXBd/lhkRd+DVBhewMQNmZ4SQuPdcD1YColg==", "title"=>"Ruby Hashes", "text"=>"\
5 Ruby Hash is a bucket with key value store."}
6 No template found for ArticlesController#create, rendering head :no_content
7 Completed 204 No Content in 76ms
```

(the above code snippet online)

What we can tell from the above is the following:

1. Rails server didn't find any view template to render for the `create` action. Which is true, because we have not created any.
2. In the absence of this view template, Rails server decided to respond with the HTTP status code `204` with response reason `No Content`.
3. The HTTP response didn't have any body that could be used for rendering. See the `head` call? It means that it returned an HTTP response with no response body.

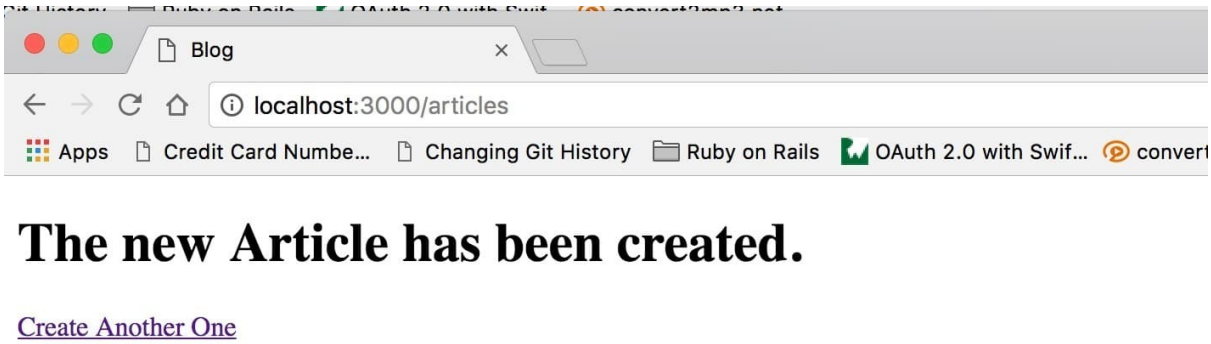
   HTTP Status Codes: We have talked a lot about these. Look for the `204` code.

However, we could have had a view template for `create` action. Let's do that. Let's create the file `app/views/articles/create.html.erb` with the following content:

```
<% # File: app/views/articles/create.html.erb %>

<h1>The new Article has been created.</h1>
<a href="/articles/new">Create Another One</a>
```

(the above code snippet online)

Now, if you submit your form, you will then be presented with the following page:



Create View Template Rendered

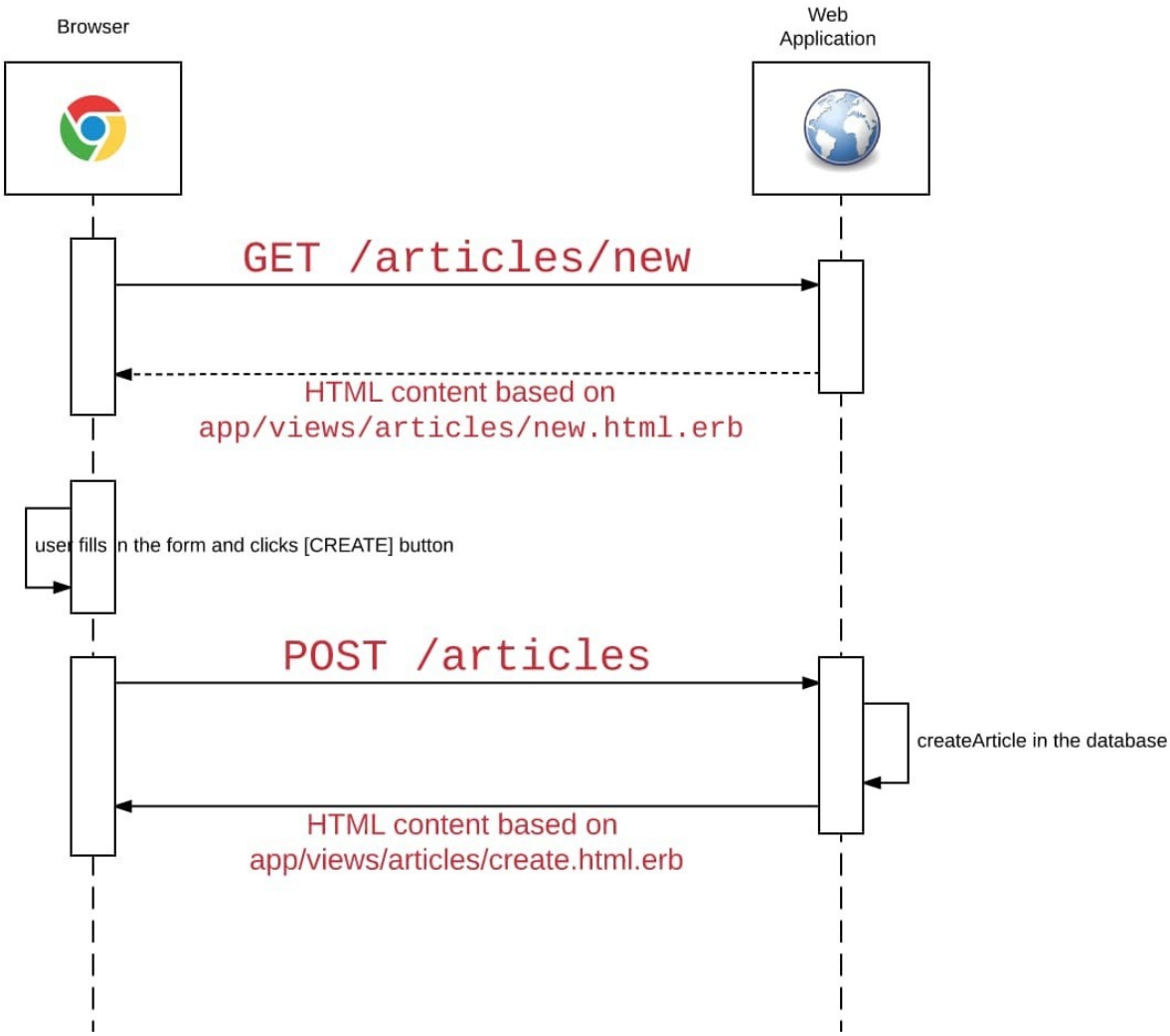
Nice! This is much more informative that our `create` action succeeded. Also, you can see the corresponding logs that now look like this:

```
Started POST "/articles" for 127.0.0.1 at 2017-05-18 23:10:43 +0300
Processing by ArticlesController#create as HTML
  Parameters: {"authenticity_token"=>"jSW02x6mO4jXUDOgzey5BY0GCISC9YyMYFX6mJhpsAO\
1EPuOZLKXBd/lhkRd+DVBhewMQNmZ4SQuPdcD1YColg==", "title"=>"Ruby Hashes", "text"=>"\
Ruby Hash is a bucket with key value store."}
  Rendering articles/create.html.erb within layouts/application
  Rendered articles/create.html.erb within layouts/application (0.5ms)
Completed 200 OK in 21ms (Views: 19.2ms)
```

(the above code snippet online)

The HTTP response was `200 OK` and we have rendered `articles/create.html.erb` within the layout `layouts/application.html.erb`.

We have finally closed the loop and we have implemented the `POST /articles` and the response back to browser:

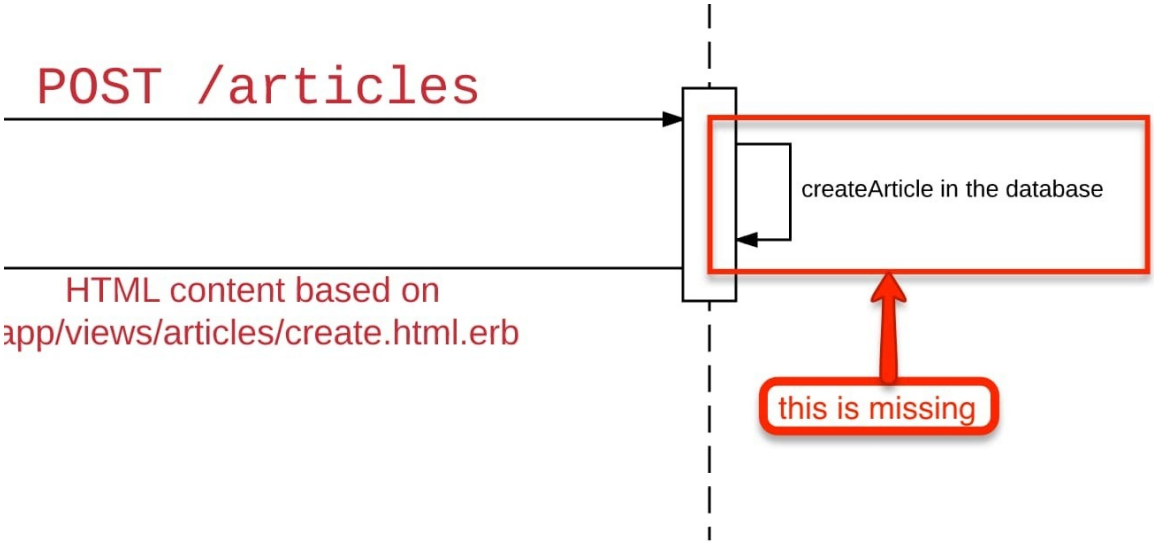


Post and then Get back the Create View Template HTML

Article Creation in Database Is Missing

Our Web application informs user that the Article has been created. But was it indeed created? Where was it created?

It's about time to bring the part of the *database* into the loop. No, the new Article was not created, because we have not bound the controller action `create` to any database creation actions.



Create Article in Database is Missing

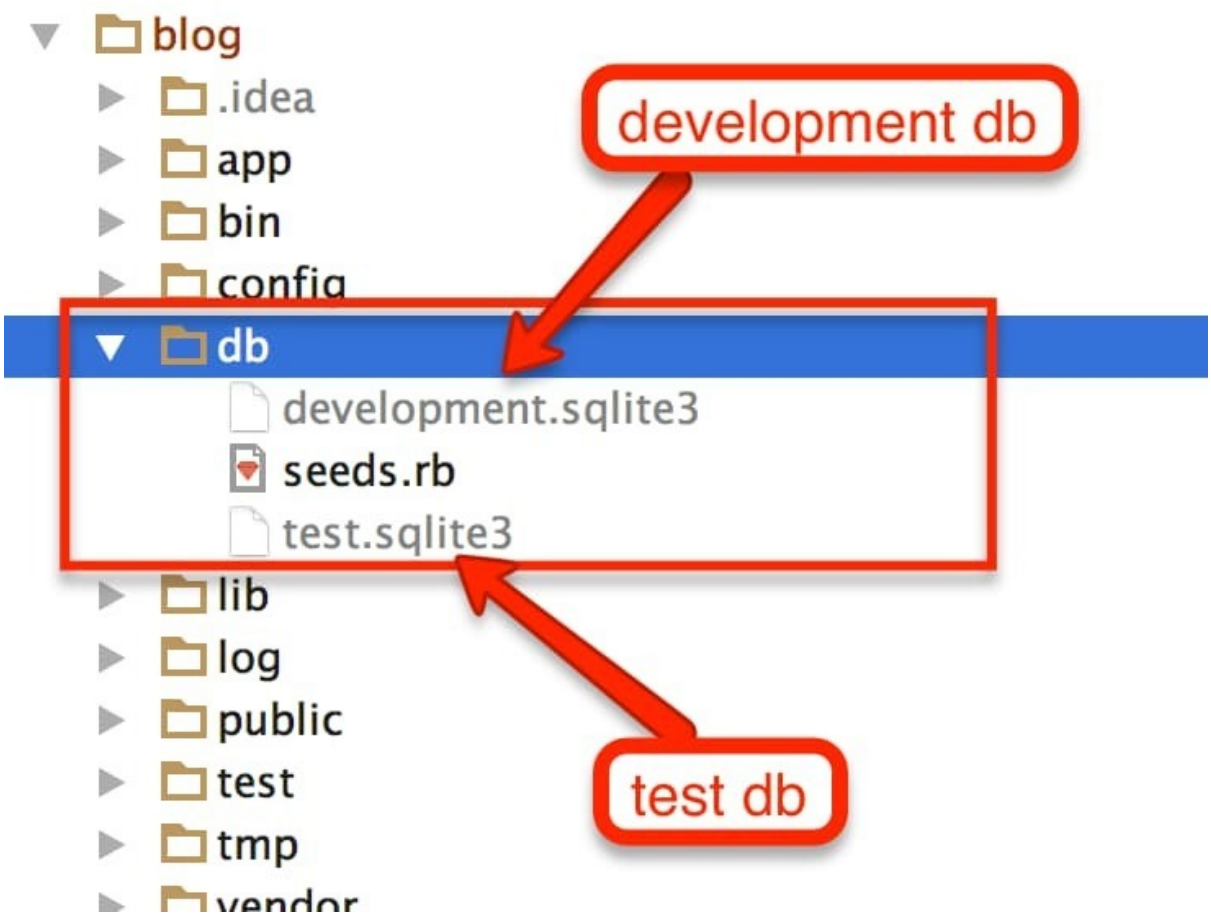
Initialize Database

Before our Web application can use the database, we need to `create` it. Let's issue the following command:

```
blog $ bin/rake db:create
Running via Spring preloader in process 27043
Created database 'db/development.sqlite3'
Created database 'db/test.sqlite3'
blog $
```

(the above code snippet online)

You can see that this command created two SQLite3 databases. One in `db/development.sqlite3` and one in `db/test.sqlite3`.



Development And Test Databases

Please, pay attention to the following. Our blog application was initialized with the command `rails new blog`. This assumed that your database was going to be an SQLite3 database. That is why now, that we start our integration with database, the `bin/rake db:create` command created two SQLite3 databases. Also, the SQLite3 databases are files that reside in the folder tree of our Web application and they are not managed by a remote database management system. If you want to read more about SQLite3 visit the page SQLite. In this course, we will work with SQLite, MySQL and PostgreSQL databases.

Create Table

Before you can save Articles into the database, you will have to create the corresponding table. Rails allows us to create tables in the database with a mechanism which is called *schema migrations*. We will spend a lot of time on these later on. For the time being, let's create a schema migration that will allow us to create the table `articles` that will be holding the Articles created by our Web application.

Run the following command on your terminal prompt:

```
blog $ bin/rails generate migration create_articles
Running via Spring preloader in process 27331
      invoke  active_record
      create    db/migrate/20170519062431_create_articles.rb
blog $
```

(the above code snippet online)

This command does not do anything to your database. But it creates a file that will allow us to tell how our `articles` table needs to be. The file that was created was `db/migrate/20170519062431_-create_articles.rb`. Note that in your case, the timestamp is going to be different. The schema migration file is a Ruby file with the following contents:

```
1 # File: db/migrate/20170519062431_create_articles.rb
2 #
3 class CreateArticles < ActiveRecord::Migration[5.1]
4   def change
5     create_table :articles do |t|
6     end
7   end
8 end
```

(the above code snippet online)

Our obligation now is to fill in the body of the block given to `create_table`, lines 5 to 6. We will add some Ruby statements that will tell Rails how our table needs to be.

Update the contents of the above file to be as follows:

```
1 # File: db/migrate/20170519062431_create_articles.rb
2 #
3 class CreateArticles < ActiveRecord::Migration[5.1]
4   def change
5     create_table :articles do |t|
6       t.string :title
7       t.text   :text
8
9       t.timestamps
10    end
11  end
12 end
```

(the above code snippet online)

You can see that we have added lines 6 till 9. These lines tell Rails that we want the table to have:

1. One column with name `title` and of type `string`.
2. One column with name `text` and of type `text`.

3. Two columns to work as create and update timestamps. The command `t.timestamps` is translated to the creation of two columns, one with name `created_at` and another one with name `updated_at` that would be automatically updated whenever the row is created and/or updated. It is a good practice to add these columns on every table that you create.

Something else that is not obvious from the above code is that Rails creates one more column in the `articles` table. It is the `id` column which is going to be the primary key of the table.

Now that we have told Rails how we want our `articles` table to be, let's ask Rails to create this table. We are going to use another `rake` task for this:

```
blog $ bin/rake db:migrate
== 20170519062431 CreateArticles: migrating ==================================
-- create_table(:articles)
   -> 0.0020s
== 20170519062431 CreateArticles: migrated (0.0020s) =========================
blog $
```

(the above code snippet online)

If everything goes ok, you will see something like the above. The schema migration has been executed and it has created the table `articles`.

We can confirm that by starting an SQLite command line interface to our `db/development.sqlite3` database. This is how you can do that. Start the SQLite command line interface by giving the following command on your project root terminal:

```
blog $ sqlite3 db/development.sqlite3
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite>
```

(the above code snippet online)

Then issue the following command to count the number of articles in your database:

```
sqlite> select count(*) from articles;
0
sqlite>
```

(the above code snippet online)

Perfect! You can leave this terminal open until we create some articles. We will use it to confirm the proper creation.

Create Model

But tables are constructs in the RDBMS world. We need to map those constructs to Ruby objects and be able to manipulate the database (create, retrieve, update, delete) using the Ruby objects instead of direct SQL. Rails offers the library which is called `ActiveRecord` which, besides many other things, it allows us to bind a Ruby class to an SQL table.

Here is where we start using the idea of the *Model*. Let's create the file `app/models/article.rb` with the following content:

```ruby
# File: app/models/article.rb
#
class Article < ApplicationRecord
end
```

(the above code snippet online)

A simple class, called `Article` that derives from another class that it is already in our project, the `ApplicationRecord` (living in the file `app/models/application_record.rb`). All `ApplicationRecord` instances will derive from `ActiveRecord::Base` which gives, for free, a lot of functionality, including ability to create, retrieve, update and delete records from the corresponding table.

Amend `create` Action

With the table and the model in place, we can now amend the `create` action to actually create the Article that the user wants to create with the submission of the form. Here is the new version of the `create` public method of the `ArticlesController`:

```ruby
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  def new
  end

  def create
    article = Article.new
    article.title = params[:title]
    article.text = params[:text]

    article.save
  end
end
```

(the above code snippet online)

Look how we have implemented the `create` method. It is very simple.

1. On line 8 we instantiate an `Article` instance.
2. On lines 9 and 10 we set its properties with the corresponding values coming in from `params`. Note that `params` hold the parameters that are coming from the HTTP request.
3. On line 12, we just call `#save` and the instance is created into the database.

   Important: This is a very simplified version of a create implementation. This should not go to production like this and we will later on learn how we should write proper, production-ready, create methods.

Look also at the Rails server logs that were output when we submitted the form:

```
Started POST "/articles" for 127.0.0.1 at 2017-05-19 14:20:19 +0300
Processing by ArticlesController#create as HTML
  Parameters: {"authenticity_token"=>"eTJy3aH5khdU5zMwrVJnB2xovhSMZhsQepuX4aHbNtN\
BBz2I2+0+mlxShtQ9RutDZIK60NcKdrg087p67DIuRg==", "title"=>"Ruby Hashes", "text"=>"\
Ruby Hash is a key/value bucket store."}
   (0.1ms)  begin transaction
  SQL (0.8ms)  INSERT INTO "articles" ("title", "text", "created_at", "updated_at\
") VALUES (?, ?, ?, ?)  [["title", "Ruby Hashes"], ["text", "Ruby Hashes"], ["cre\
ated_at", "2017-05-19 11:20:19.789470"], ["updated_at", "2017-05-19 11:20:19.7894\
70"]]
   (0.4ms)  commit transaction
  Rendering articles/create.html.erb within layouts/application
  Rendered articles/create.html.erb within layouts/application (0.6ms)
Completed 200 OK in 39ms (Views: 19.5ms | ActiveRecord: 1.9ms)
```

(the above code snippet online)

1. Do you see the `Hash` with the parameters? The hash you see logged is the one that you can access with the `params` method. Do you see that it has the `title` and `text` keys with the values being the values we used to fill in the form for Article creation?
2. Do you see the SQL statement that was automatically generated and executed? `INSERT INTO "articles" .....` This is the magic of `ActiveRecord`.

Note that on lines 9 and 10 in the `create` method we access the columns of the table `articles` with corresponding setter methods in the `Article` model. Even if we have not declared these setter methods, `ActiveRecord` does this for us automatically by inspecting on the table `articles` definition. Looking into its definition it can find out which are the columns and allow us to use the corresponding setter and getter methods.

Confirm on SQLite Command Line

And we can now confirm that the article has been created into the database, using the SQLite command line interface:

```
sqlite> select count(*) from articles;
1
sqlite> select * from articles;
1|Ruby Hashes|Ruby Hashes|2017-05-19 11:20:19.789470|2017-05-19 11:20:19.789470
sqlite>
```

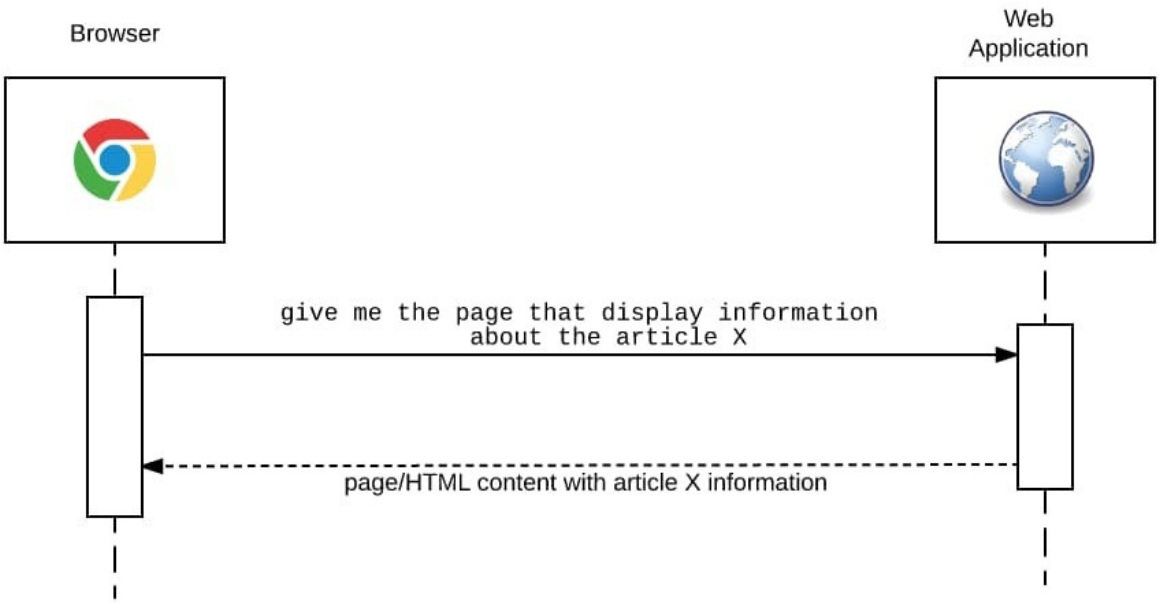
(the above code snippet online)

> *Side Note:* Earlier we have used two of the most popular commands in Rails development:
>
> 1. `bin/rake ....`
> 2. `bin/rails generate ....`
>
> Many batch type commands are provided by Rake tasks and you can see all the available rake tasks if you run `bin/rake --tasks`. On the other hand, the `bin/rails generate` is another type of commands that usually generate Ruby code for you. You can invoke the command `bin/rails generate` in order to get the list of all the available generators. We have used the `migration` one.

Retrieve - (R)

The above closes the Create part of the *management* of a resource, like the `Article`. Now that we have an Article created, another management action that we need to have a look at is the *retrieve (R)* action.

With the retrieve, we want to fetch a particular resource and display its information to the user. This is what we usually do when we fetch a Web page, and as you can imagine, we are talking here about another `GET` case.



Client Requests The Page of a Particular Article

Routes

First thing that we have to do is to amend our routes in order to expose an endpoint that would refer to a specific article. Here is how we do that:

```
1 # File: config/routes.rb
2 #
3 Rails.application.routes.draw do
4   get 'articles/new'
5   post 'articles', to: 'articles#create'
6   get 'articles/:id', to: 'articles#show'
7 end
```

(the above code snippet online)

Look at line number 6. `get 'articles/:id', to: 'articles#show'`. The `:id` is a placeholder that would be used to refer to a specific article. It is the responsibility of the `show` action that will handle the request, to take the actual value and fetch the correct article from the database. In other words, if the user requests `/articles/15`, then the `show` action will have to retrieve the article with id `15` and make sure it returns back a page that displays the information for this particular article. If the user requests `/articles/1`, then `show` will fetch details for article with id `1`, and so on.

You can confirm the routes table if you run:

```
1 blog $ bin/rake routes
2 Running via Spring preloader in process 32096
3       Prefix Verb URI Pattern             Controller#Action
4 articles_new GET  /articles/new(.:format) articles#new
5     articles POST /articles(.:format)     articles#create
6              GET  /articles/:id(.:format) articles#show
7 blog $
```

(the above code snippet online)

You can see the last `GET /articles/:id` entry that takes us to the `articles` controller `show` action.

Show Action

With the above routes in place, let's visit the URL of an article. If you see the contents of the table `articles`, you will see that the single article in, it has the id with value `1`:

```
1 sqlite> select * from articles;
2 1|Ruby Hashes|Ruby Hashes|2017-05-19 11:20:19.789470|2017-05-19 11:20:19.789470
3 sqlite>
```

(the above code snippet online)

Look at the first column. But if your article has another id, then use that one instead.

Visit the page http://localhost:3000/articles/1. What do you see? You get an error that the `show` action does not exist in the Articles controller. Which was expected.

Let's implement the `show` action. This is the new version of the `ArticlesController`:

```
1  # File: app/controllers/articles_controller.rb
2  #
3  class ArticlesController < ApplicationController
4    def new
5    end
6
7    def create
8      article = Article.new
9      article.title = params[:title]
10     article.text = params[:text]
11
12     article.save
13   end
14
15   def show
16     @article = Article.find(params[:id])
17   end
18 end
```

(the above code snippet online)

You can see the new lines, 15 till 17. The implementation is very simply. We are using the `ActiveRecord #find` method, which takes as input an id and tries to locate the row with the given id in the database. See how `params` is holding the value of the `:id` part of the URI path, which is pretty cool!

Note also that we save the result into an instance variable `@article`. This is necessary because we will use the instance variable to fill in the dynamic information of the corresponding view later on. We used the same technique when we worked with Sinatra in the Sinatra chapter.

Show View

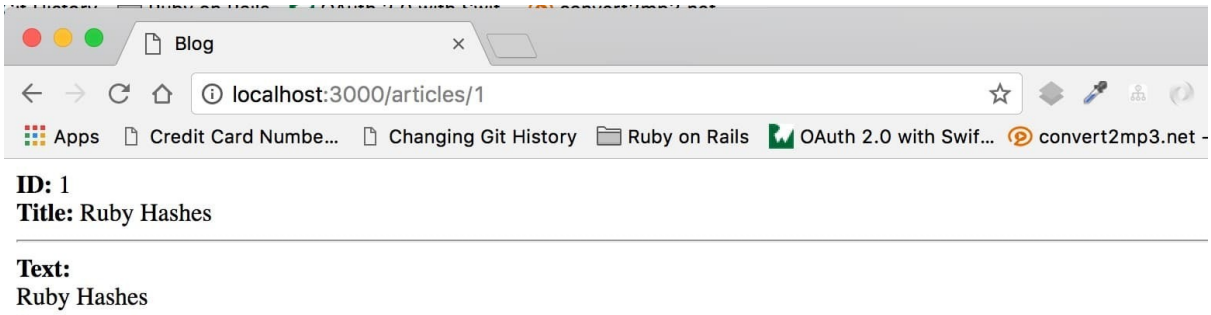
If we refresh the page http://localhost:3000/articles/1, we will now get the error that the `show` action is missing a view template. Let's create the file `app/views/articles/show.html.erb`.

```
<% # File: app/views/articles/show.html.erb %>

<div>
  <strong>ID:</strong> <%= @article.id %>
</div>

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

<hr/>

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

(the above code snippet online)

This is pretty simple. Do you see the dynamic parts? These are the parts that are built with the help of ERB code. Inside the ERB blocks, we use the instance variable `@article` as this is the variable that has been instantiated by the `show` action of the `ArticlesController`. So, `@article.id` is used to print the id of the Article, `@article.title` is used to print the title of the Article and, finally, `@article.text` is used to print the text of the article.

Now, if you refresh the page on your browser, you will get something like this:



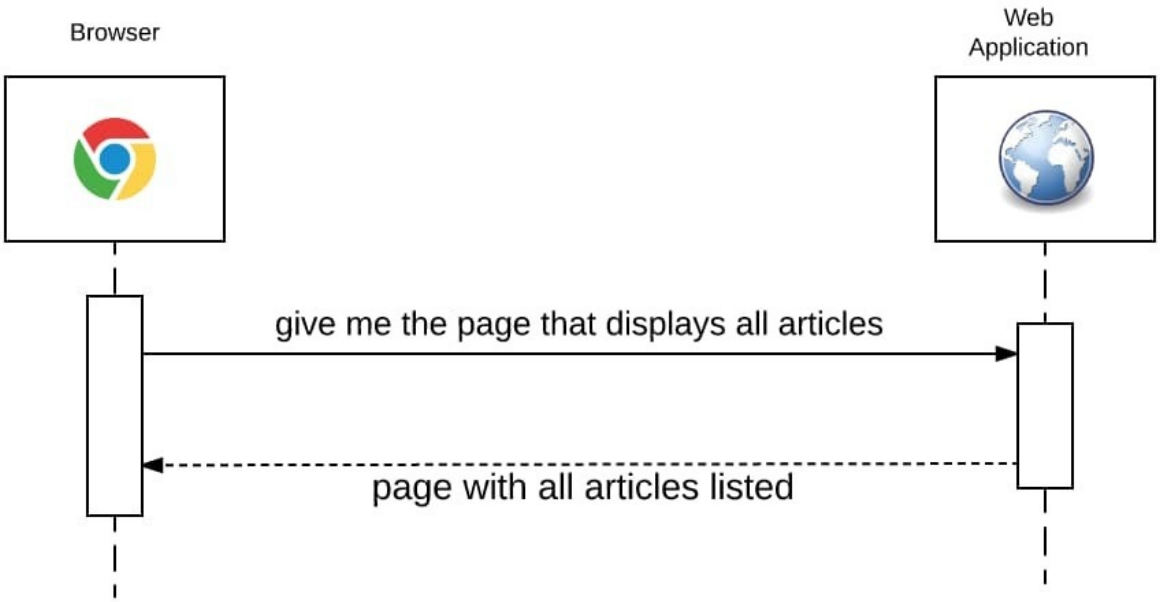
Show Page Displays the Details Of The Article Requested

Nice! And this closes the loop for the retrieve (R) management action.

Client Sends GET Request For Specific Article

Retrieve Many

But, besides the action to retrieve 1 specific resource, we may equally well want to retrieve many resources of the same type. For example, we might want to retrieve all the Articles in our database.



Browser Wants To Get the Page With All Articles Listed

Routes

Obviously, we need a `GET` request again. But what is going to be the URI to refer to all articles. This is very easy: `/articles`. Let's add this endpoint inside our routes file.

```
1 # File: config/routes.rb
2 #
3 Rails.application.routes.draw do
4   get 'articles/new'
5   post 'articles', to: 'articles#create'
6   get 'articles/:id', to: 'articles#show'
7   get 'articles', to: 'articles#index'
8 end
```

(the above code snippet online)

It is the line 7 that we have added. The `get 'articles'` HTTP endpoint will be handled by the `articles` controller and its action `index`. This is the Rails convention for this kind of endpoints.

Implement `index` Action

Let's now implement the `index` action.

```ruby
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  def new
  end

  def create
    article = Article.new
    article.title = params[:title]
    article.text = params[:text]

    article.save
  end

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

  def index
    @articles = Article.all
  end
end
```

(the above code snippet online)

We have added lines 19 to 21. The implementation is very simple. We call `Article.all`, which is a way to access all the Articles in the database and we save the result into an instance variable with name `@articles`. Again, we need the instance variable in order to use it in the corresponding `index` view that will build the HTML content for the response back to the client browser.

View Template for `index` Action

Let's now add the view template for the `index` action of the articles controller. Create the file `app/views/articles/index.html.erb` with the following content:

```
<% # File: app/views/articles/index.html.erb %>

<h1>List Of Articles</h1>

<table>
  <thead>
    <tr><th>ID</th><th>Title</th><th>Text</th></tr>
  </thead>
  <tbody>
    <% @articles.each do |article| %>
      <tr><td><%= article.id %></td><td><%= article.title %></td><td><%= article.\
text %></td></tr>
```

```
    <% end %>
  </tbody>
</table>
```

(the above code snippet online)

Using the instance variable `@articles`, we use ERB to iterate over the collection of articles. For each one of the articles, which is referenced by the block variable `article`, we print an HTML table row `<tr>...</tr>` with the details of the article. See how we use snippets of ERB to print the id, the title and the text of the article at hand, as content in the corresponding table `<td>...</td>` HTML elements.

And this closes the loop for the retrieve many resources interaction.

Browser Calls GET /articles and Gets Back What index View Renders

Add some Links To Help Navigation

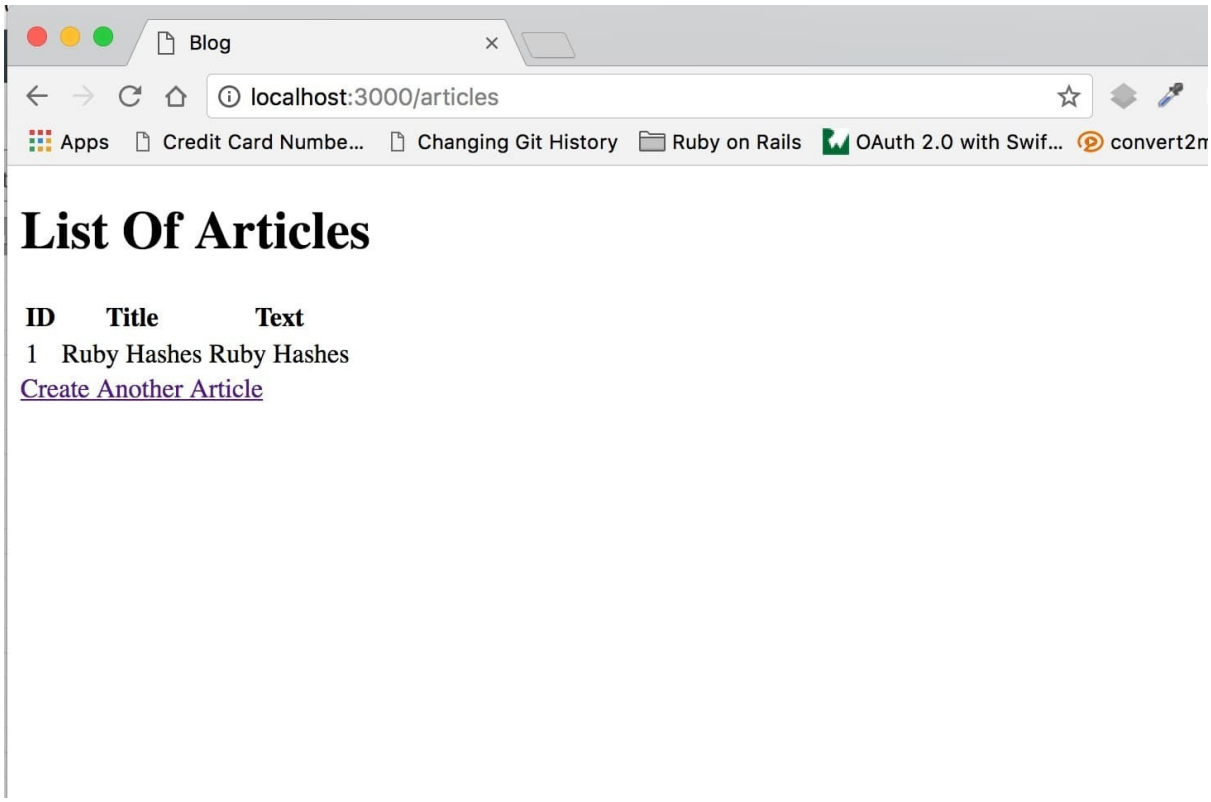
Before we go to the next management action, the *update*, let's add one link to the `index` view that would allow us to create a new article.

Let's add the following at the end of the `app/views/articles/index.html.erb` file:

```
...
<div>
  <a href="/articles/new">Create Another Article</a>
</div>
```

(the above code snippet online)

If you load the list of articles page, http://localhost:3000/articles, you will see this:



List Of Articles Page

You can see the `Create Another Article` link that would allow you to create more Articles.

Also, add a link to list of articles from the `create` view:

```
1 <% # File: app/views/articles/create.html.erb %>
2
3 <h1>The new Article has been created.</h1>
4 <a href="/articles/new">Create Another One</a>
5 <div>
6   <a href="/articles">List of Articles</a>
7 </div>
```

(the above code snippet online)

Do you see lines 5 till 7? This is a new link that would take you to the list of Articles.

Add the same link inside the page that renders the form to add a new Article:

```
<% # File: app/views/articles/new.html.erb %>

<h1>Create a New Article</h1>
<form method="post" action="/articles">
  <input type="hidden" id="authenticity-token" name="authenticity_token" value="<\
%= form_authenticity_token %>"/>
  <div>
    <label for="title">Title:</label>
    <input type="text" id="title" name="title" placeholder="Give Title of Article\
"/>
  </div>

  <div>
    <label for="text">Text:</label>
    <textarea id="text" name="text" placeholder="Type in the text of the Article"\
></textarea>
  </div>

  <div>
    <button type="submit">Create Article</button>
  </div>
</form>

<div>
  <a href="/articles">List of Articles</a>
</div>
```

(the above code snippet online)

Also, add a link to the show page on each Article entry in the `index` view:

```
<% # File: app/views/articles/index.html.erb %>

<h1>List Of Articles</h1>

<table>
  <thead>
    <tr><th>ID</th><th>Title</th><th>Text</th><th></th></tr>
  </thead>
  <tbody>
    <% @articles.each do |article| %>
      <tr><td><%= article.id %></td><td><%= article.title %></td><td><%= article.\
text %></td><td><a href="/articles/<%= article.id%>">Show</a></td></tr>
    <% end %>
  </tbody>
</table>
<div>
```

```
17   <a href="/articles/new">Create Another Article</a>
18 </div>
```

(the above code snippet online)

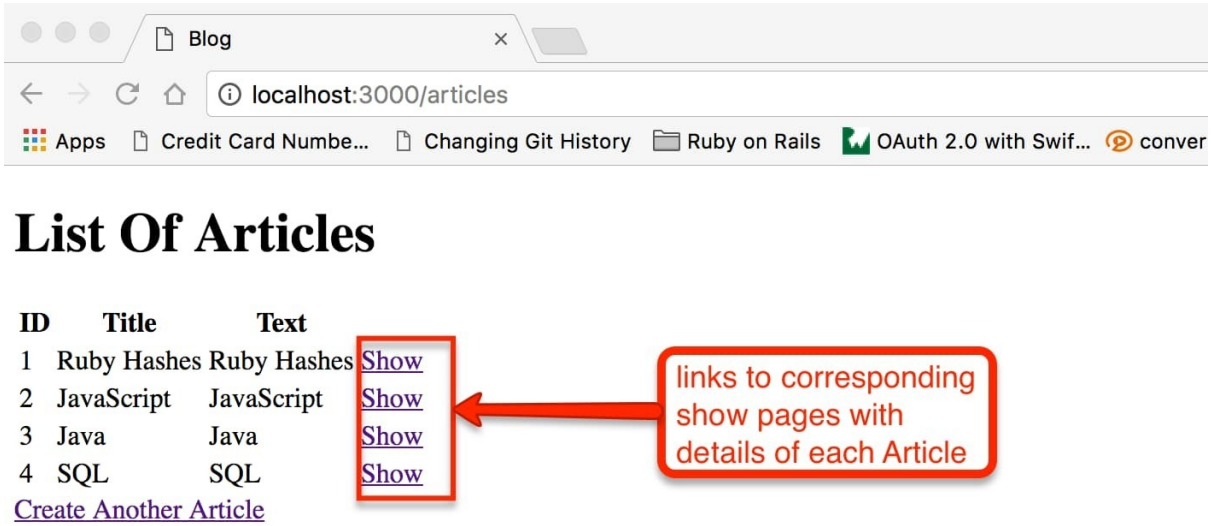
Look how we have added the new cell content on line 11:

```
1 <td><a href="/articles/<%= article.id%>">Show</a></td>
```

(the above code snippet online)

We use the `article.id` and ERB to create the href dynamically.

Now, if you load the list of articles page, you will see a <u style="color: blue;">Show</u> link on every row on every Article. Clicking on this link will take you to the corresponding show page.



Links To Show Pages

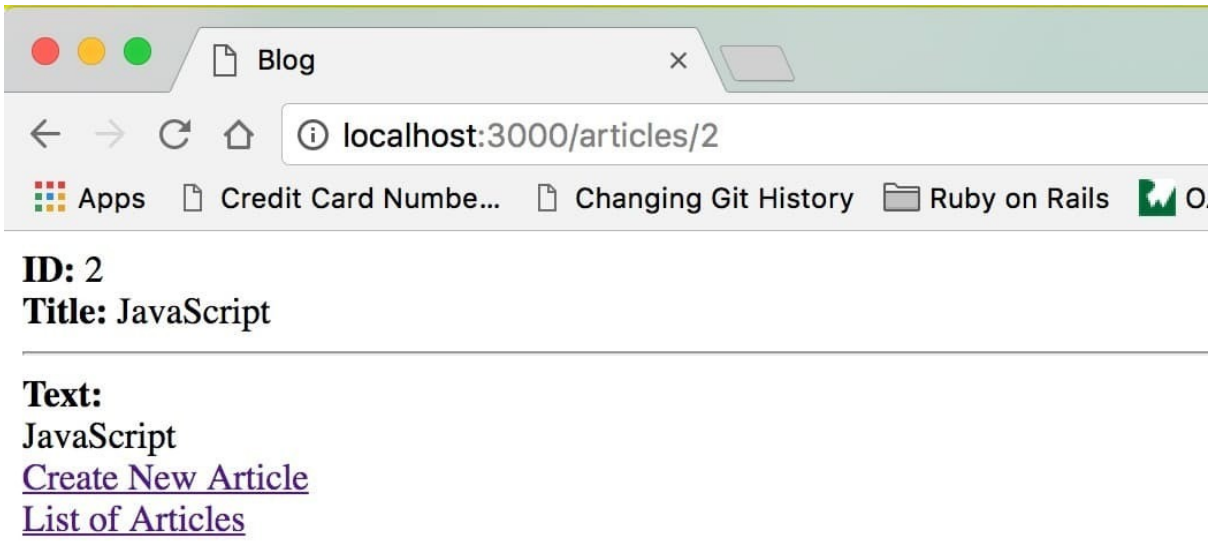
Finally, add two links inside the show view template. One that would take you to the create new article page and one that would take you to the list of articles page:

```
1 <% # File: app/views/articles/show.html.erb %>
2
3 <div>
4   <strong>ID:</strong> <%= @article.id %>
5 </div>
6
7 <div>
8   <strong>Title:</strong> <%= @article.title %>
9 </div>
10
11 <hr/>
12
```

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

<div>
  <a href="/articles/new">Create New Article</a>
</div>

<div>
  <a href="/articles">List of Articles</a>
</div>
```

(the above code snippet online)

Now, visiting a show page will look like this:



Show Page With Links To Create and List Pages

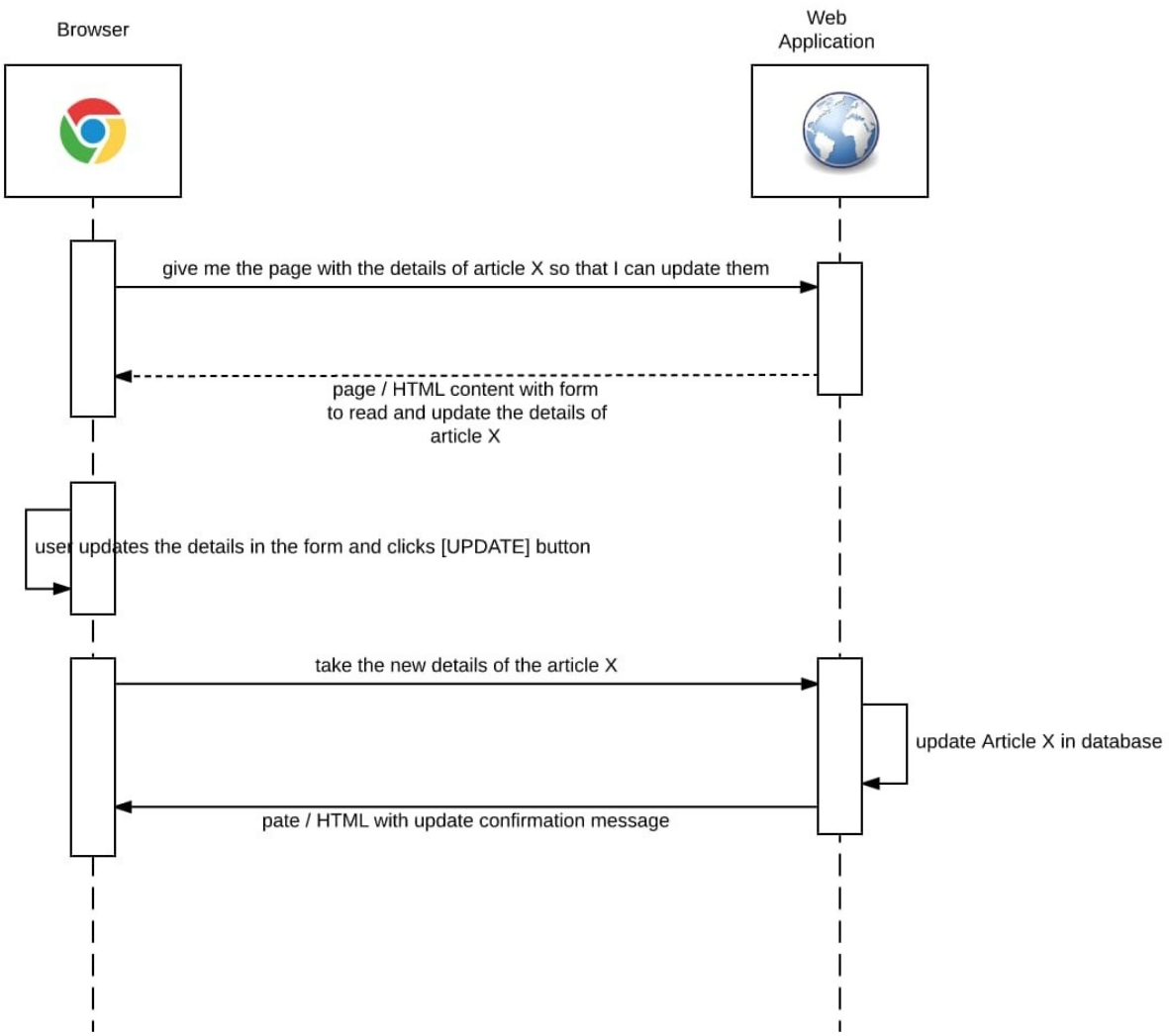
Now our application has the basic navigation links that are related to the functionality that we have already added, i.e. create, retrieve and retrieve many.

Update (U)

Now it's about time to allow for updates of the resources, i.e. for an update of a particular article. Updates work similar to create:

1. We need a page with a form. This form will display the details of the Article we are updating. It will allow us to update those details and submit the changes in order for the Article to be updated into the database.

2. We need a page that will display a confirmation that the update has been executed successfully.



Browser Exchanges Messages With Server To Update Specific Article

Routes

As earlier, we need to specify an endpoint that will allow us to fetch the page with the form to edit a specific resource, i.e. a specific Article. What about `GET /articles/:id/edit`? It specifies the verb as `GET` since this request is only fetching from server and does not change any of the state of the resource. Also, it has a placeholder for the specification of the id of the Article that we want to edit. And finally, the `edit` suffix in the path is necessary in order to avoid confusion with the show endpoint (`GET /articles/:id` is used for displaying the details of an Article).

Let's add the entry in the routes.

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  get 'articles/new'
  get 'articles/:id/edit', to: 'articles#edit'
  get 'articles/:id', to: 'articles#show'
  get 'articles', to: 'articles#index'
  post 'articles', to: 'articles#create'
end
```

(the above code snippet online)

The new line that we have added is line 5: `get 'articles/:id/edit', to: 'articles#edit'`. Let's run the command that will print the routes:

```
blog $ bin/rake routes
Running via Spring preloader in process 42038
      Prefix Verb URI Pattern                  Controller#Action
articles_new GET  /articles/new(.:format)      articles#new
             GET  /articles/:id/edit(.:format) articles#edit
             GET  /articles/:id(.:format)      articles#show
    articles GET  /articles(.:format)          articles#index
             POST /articles(.:format)          articles#create
blog $
```

(the above code snippet online)

You can now see the entry `GET /articles/:id/edit(.:format) articles#edit` which is the result of the line that we have just added.

`edit` Action

The `edit` action needs to return back the content for the user to edit the details of the article specified. Let's implement this action.

```ruby
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  def new
  end

  def create
    article = Article.new
    article.title = params[:title]
    article.text = params[:text]

    article.save
```

```
13    end
14
15    def show
16      @article = Article.find(params[:id])
17    end
18
19    def index
20      @articles = Article.all
21    end
22
23    def edit
24      @article = Article.find(params[:id])
25    end
26  end
```

(the above code snippet online)

We have added the lines 23 to 25. It is a very simple implementation. Like in the `show` action we just locate the Article given its id. We instantiate the `@article` instance variable that will be used in the corresponding view template for the `edit` action.

`edit` View Template

Now, we have already learned that we need an `edit` view template. Let's create the file `app/views/articles/edit.html.erb` with the following content:

```
1  <% # File: app/views/articles/edit.html.erb %>
2
3  <h1>Update Article with ID: <%= @article.id %></h1>
4
5  <form method="post" action="/articles/<%= @article.id %>">
6    <input type="hidden" id="authenticity-token" name="authenticity_token" value="<\
7  %= form_authenticity_token %>"/>
8    <input type="hidden" name="_method" value="put"/>
9    <div>
10     <label for="title">Title:</label>
11     <input type="text" id="title" name="title" placeholder="Give Title of Article\
12 " value="<%= @article.title %>"/>
13   </div>
14
15   <div>
16     <label for="text">Text:</label>
17     <textarea id="text" name="text" placeholder="Type in the text of the Article"\
18 ><%= @article.text %></textarea>
19   </div>
20
21   <div>
```

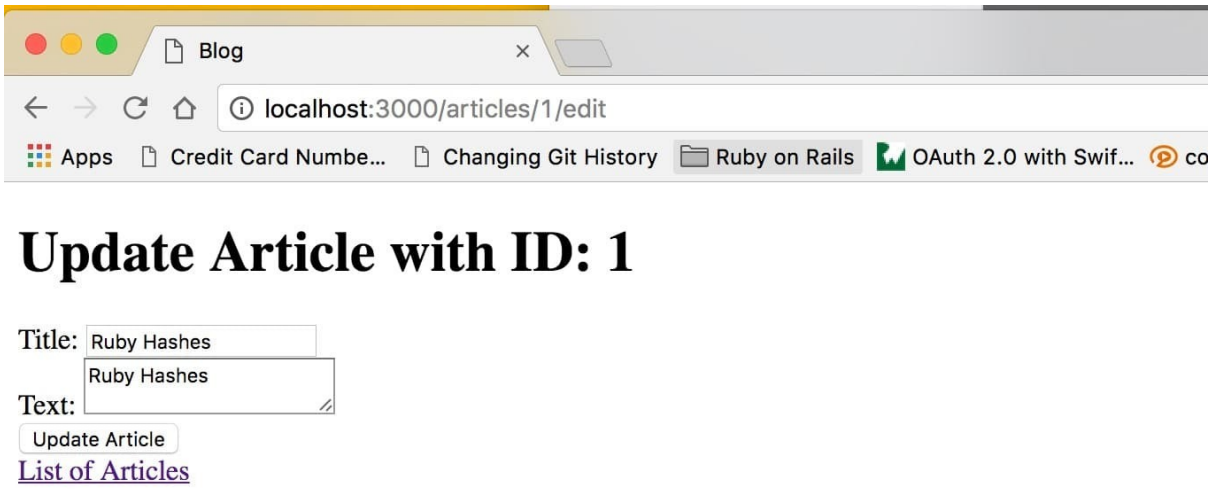
```
22      <button type="submit">Update Article</button>
23    </div>
24  </form>
25
26  <div>
27    <a href="/articles">List of Articles</a>
28  </div>
```

(the above code snippet online)

The above, is almost the same like the `new.html.erb` view template. However, there are some things that you need to be careful with.

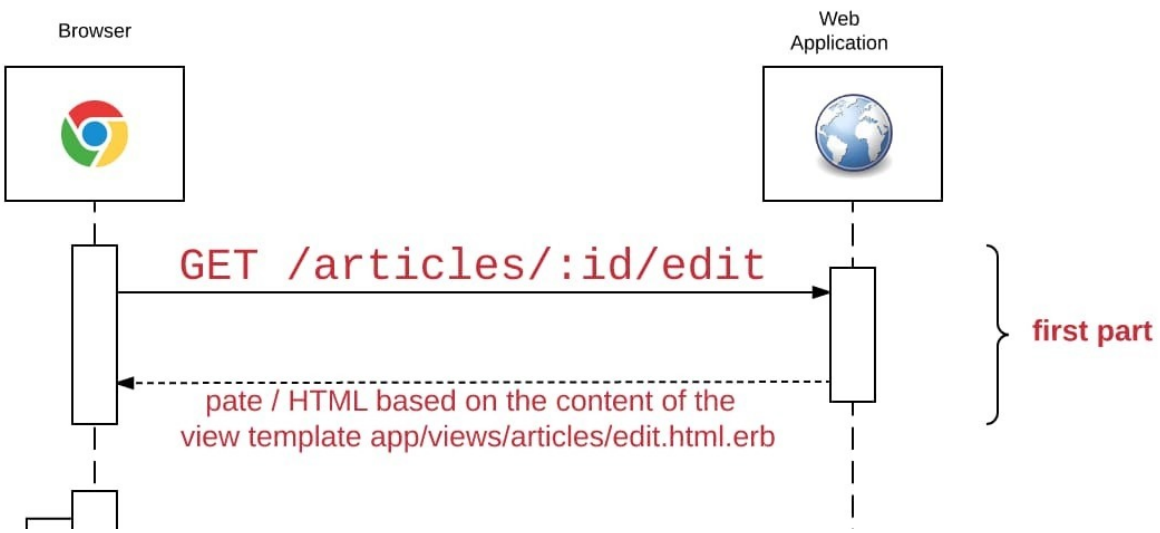
1. Although the method declared at the `form` level is `post`, we use a `hidden` input field with name `_method` and value `put`. This is to tell Rails that we want our `POST /articles/:id` request to be handled by the `PUT /articles/:id` endpoint instead. Why did we do that? Why don't we just put `method="put"` at the form HTML tag? This is because the `method="put"` is not supported by all browsers and if we use it the browser just sends a `GET` request and not a `PUT` request. In order to ask the browser to send form data to Rails server and still handle the HTTP request as a `PUT` request, we use this trick with the `hidden` input field.
2. The `action` has the value `/articles/:id` because we need to tell Rails server which article we are interested to update.
3. The input fields now have values that they start with (line 10 and line 15). The values are taken from the values of the properties of the `@article` instance variable that the `edit` action instantiates.

Cool! Try visiting the URL http://localhost:3000/articles/1/edit. Thus will open the page:



Update Page for Article with ID 1

And this closes the first part of the update feature. It is the request for the page to fill in the new details of a specific article.



Browser Gets the Page To Edit Specific Article

Let's Add Links To Edit Page

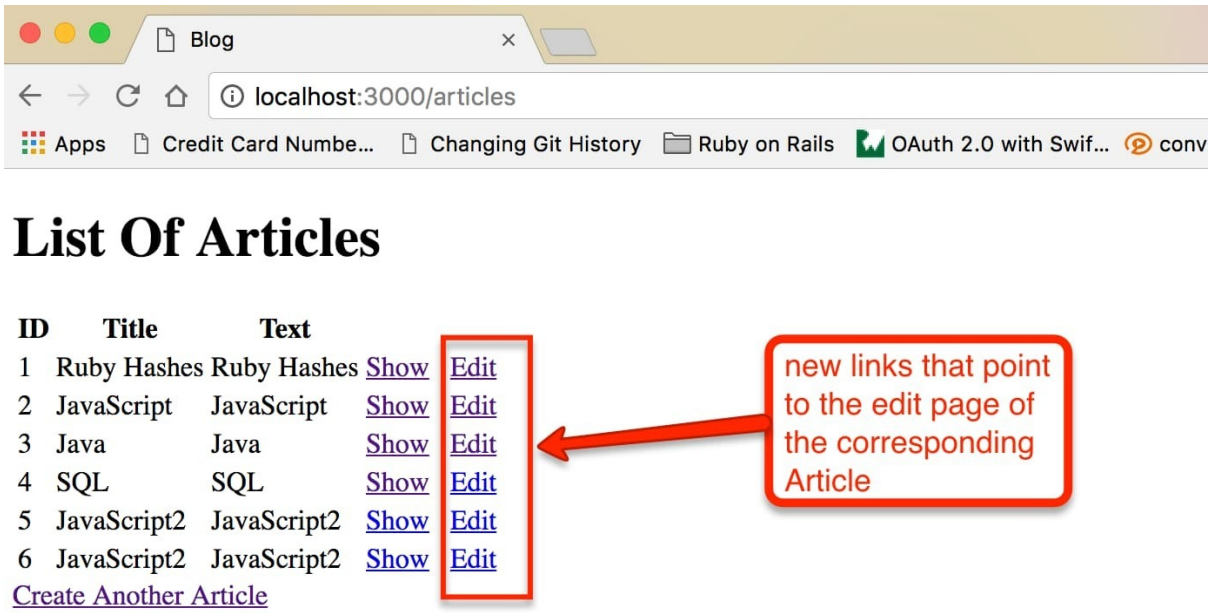
Before we continue with the second part of the page, let's add some links that will help us navigate to the edit pages of the articles.

Firstly, let's add edit links next to each <u style="color: blue;">Show</u> link in the page with the list of all articles. Here is the new version of the page:

```
1  <% # File: app/views/articles/index.html.erb %>
2
3  <h1>List Of Articles</h1>
4
5  <table>
6    <thead>
7      <tr><th>ID</th><th>Title</th><th>Text</th><th></th></tr>
8    </thead>
9    <tbody>
10     <% @articles.each do |article| %>
11       <tr>
12         <td><%= article.id %></td>
13         <td><%= article.title %></td>
14         <td><%= article.text %></td>
15         <td>
16           <a href="/articles/<%= article.id%>">Show</a>  
17           <a href="/articles/<%= article.id%>/edit">Edit</a>
18         </td>
19       </tr>
20     <% end %>
21   </tbody>
22 </table>
23 <div>
24   <a href="/articles/new">Create Another Article</a>
25 </div>
```

(the above code snippet online)

Besides some code styling changes, the only new entry is on line 17. There is a new anchor that points to the edit page of the corresponding Article. Hence, if you load the list of Articles page with some articles already in, you will see something like this:



List Of Articles Page With Links To Edit Pages

Secondly, let's put an edit link inside the show page of each Article. Here is the new version of the `app/views/articles/show.html.erb`:

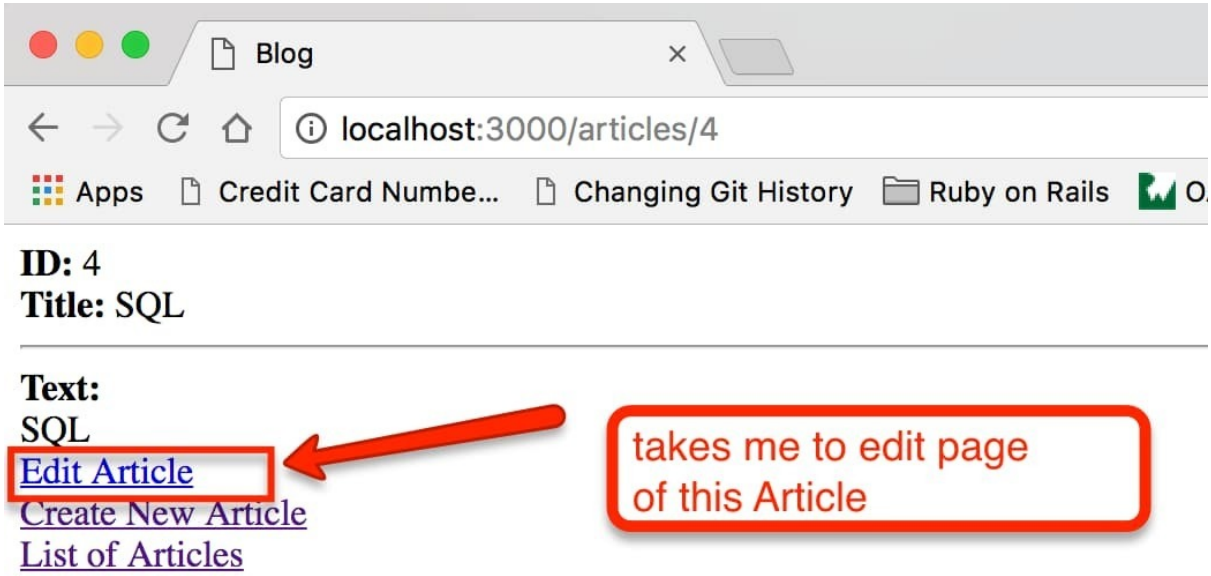
```
<% # File: app/views/articles/show.html.erb %>

<div>
  <strong>ID:</strong> <%= @article.id %>
</div>

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

<hr/>

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

<div>
  <a href="/articles/<%= @article.id %>/edit">Edit Article</a>
</div>

<div>
  <a href="/articles/new">Create New Article</a>
</div>
```

```
25
26 <div>
27   <a href="/articles">List of Articles</a>
28 </div>
```

(the above code snippet online)

You can see the new link inserted in lines 18 to 20.

Now, if you load the show page of an Article, you can see that it has a link to take you to its corresponding edit page:



Show Page Has Edit Link

Submit Data For Update

With the navigation links in place, let us now implement the second part of the update. We need to send the form data of the edit page back to the Rails server. And the Rails server needs to process it and return back the update confirmation page.

Do we have the correct endpoint set up? The route? We said that the form in edit view template will send a request to `/articles/:id`. Let's add the correct entry in `config/routes.rb` file:

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  get 'articles/new'
  get 'articles/:id/edit', to: 'articles#edit'
  get 'articles/:id', to: 'articles#show'
  get 'articles', to: 'articles#index'
  post 'articles', to: 'articles#create'
  put 'articles/:id', to: 'articles#update'
end
```

(the above code snippet online)

It is the last line that we have added, line 9. It will be the endpoint for `PUT /articles/:id` and it will be handled by the `update` action of the Articles controller. Let's run the command that will print all the routes that we now have in our configuration.

```
blog $ bin/rake routes
Running via Spring preloader in process 6313
      Prefix Verb URI Pattern                  Controller#Action
articles_new GET  /articles/new(.:format)      articles#new
             GET  /articles/:id/edit(.:format) articles#edit
             GET  /articles/:id(.:format)      articles#show
    articles GET  /articles(.:format)          articles#index
             POST /articles(.:format)          articles#create
             PUT  /articles/:id(.:format)      articles#update
blog $
```

(the above code snippet online)

Do you see the last line printed? It is the one generated by the `config/routes.rb` line that we have just added.

`update` Action

Let's now implement the `update` action in Articles controller:

```ruby
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  def new
  end

  def create
    article = Article.new
    article.title = params[:title]
    article.text = params[:text]
```

```
11
12      article.save
13    end
14
15    def show
16      @article = Article.find(params[:id])
17    end
18
19    def index
20      @articles = Article.all
21    end
22
23    def edit
24      @article = Article.find(params[:id])
25    end
26
27    def update
28      @article = Article.find(params[:id])
29      @article.title = params[:title]
30      @article.text = params[:text]
31
32      @article.save
33    end
34  end
```

(the above code snippet online)

Look at the lines 27 to 33. This is the implementation of the `update` action. It is pretty simple. It locates the article specified by the `:id` parameter and then updates its properties according to the values of the parameters `:title` and `:text`.

Before we try this on our browser, let's also add the `app/views/articles/update.html.erb` file that will be the confirmation page that the update has been completed.

`update` View Template

Create the file `app/views/articles/update.html.erb` with the following content:

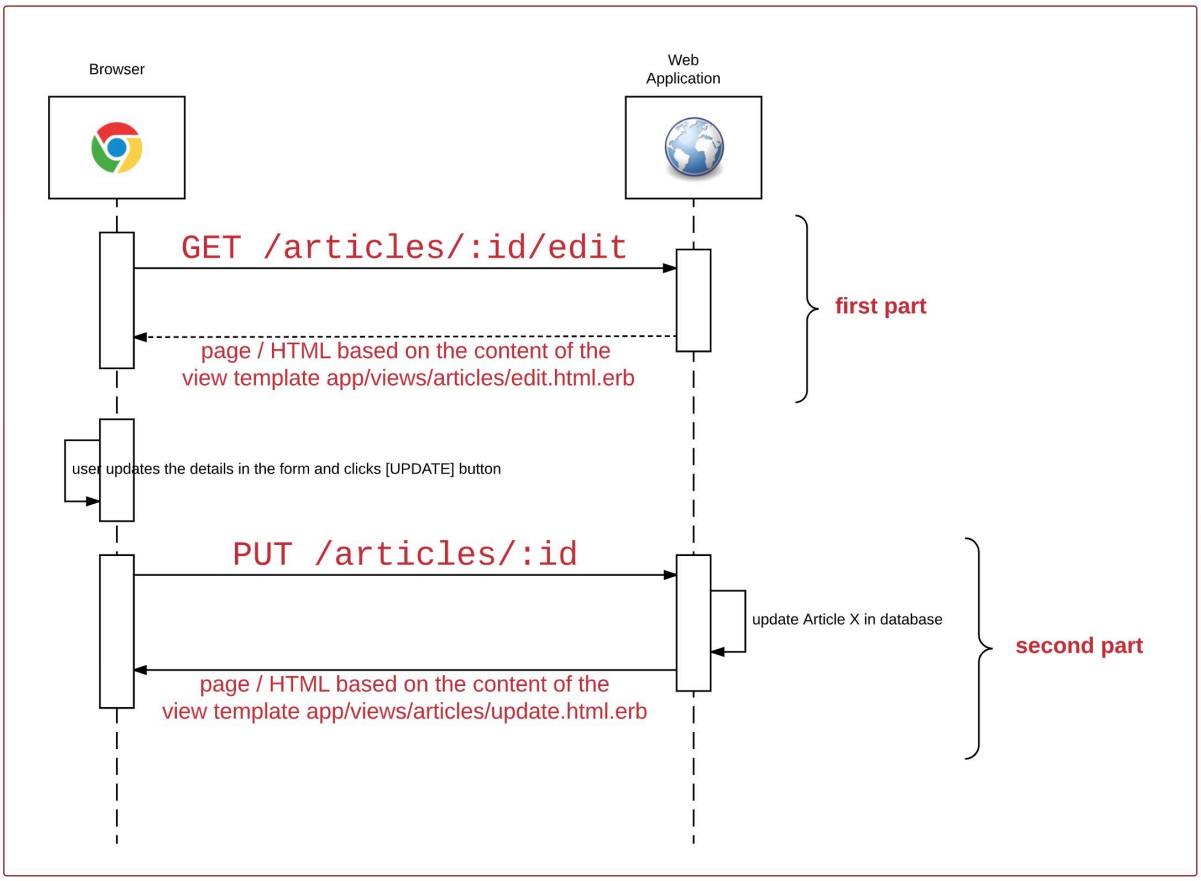
```
1  <% # File: app/views/articles/update.html.erb %>
2
3  <h1>Article with ID <%= @article.id %> has been updated</h1>
4
5  <div>
6    <a href="/articles/<%= @article.id %>">View Article Details</a>
7  </div>
8
9  <div>
```

```
  <a href="/articles">List Of Articles</a>
</div>
```

(the above code snippet online)

This is a very simple page that informs us that the update has been carried out. Also, it has a link to visit the show page of the article that has been updated. And another link to visit the list of articles page.

Now, all the functionality to update is there. We have implemented the second part too:



Edit Update Interaction Diagram

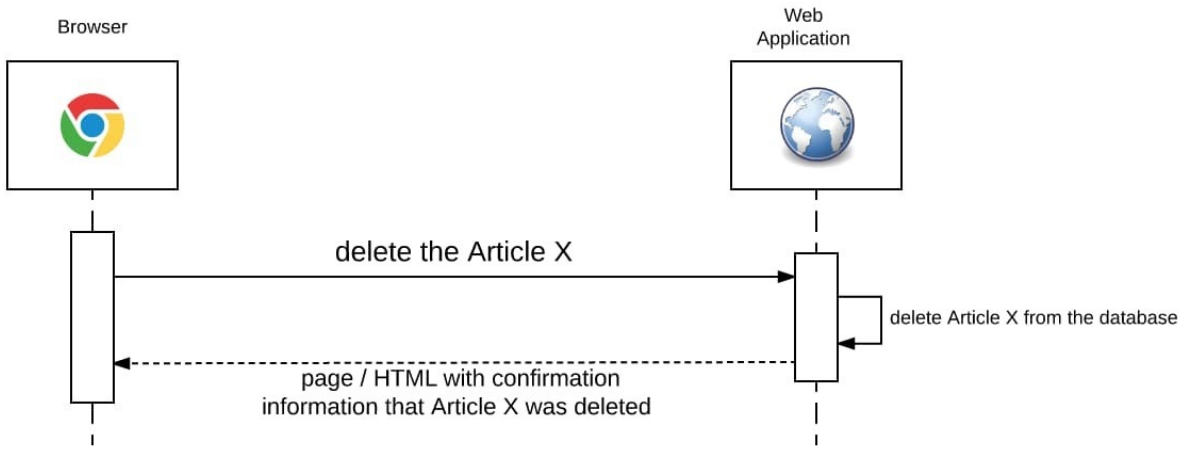
Try it on your browser. Make sure that you also watch the Rails server logs. You will learn a lot by just reading the logs.

The following video shows, in action, the whole functionality, that we have built so far.

Blog Demo Articles Management First Part

Delete (D)

There is a last action that we need to implement. That of the deletion of an Article.



Client Wants To Delete an Article

The client, i.e. browser, will need to send a request to Rails server specifying that they want a specific Article to be deleted. Then, Rails Server will delete the article and return back a Web page with confirmation on deletion.

Route

Firstly, we need to specify the endpoint that Rails will accept Article deletion requests. Let's update our routes file:

```
1  # File: config/routes.rb
2  #
3  Rails.application.routes.draw do
4    get 'articles/new'
5    get 'articles/:id/edit', to: 'articles#edit'
6    get 'articles/:id', to: 'articles#show'
7    get 'articles', to: 'articles#index'
8    post 'articles', to: 'articles#create'
9    put 'articles/:id', to: 'articles#update'
10   delete 'articles/:id', to: 'articles#destroy'
11 end
```

(the above code snippet online)

It is line 10 that we have added. We tell Rails to accept HTTP verb `DELETE` on `/articles/:id` URI. We also tell it to handle this HTTP request with the action `destroy` of the Articles controller. Why `destroy` and not `delete` This is a convention that Rails uses for public controller actions that delete a resource.

`destroy` Action

So, let's now implement the `destroy` action in the Articles controller:

```
1  # File: app/controllers/articles_controller.rb
2  #
3  class ArticlesController < ApplicationController
4    def new
5    end
6
7    def create
8      article = Article.new
9      article.title = params[:title]
10     article.text = params[:text]
11
12     article.save
13   end
14
15   def show
16     @article = Article.find(params[:id])
17   end
18
19   def index
20     @articles = Article.all
21   end
22
23   def edit
24     @article = Article.find(params[:id])
25   end
26
27   def update
28     @article = Article.find(params[:id])
29     @article.title = params[:title]
30     @article.text = params[:text]
31
32     @article.save
33   end
34
35   def destroy
36     article = Article.find(params[:id])
37     article.destroy
38   end
39 end
```

(the above code snippet online)

The implementation of the `destroy` action, lines 35 to 38, is very simple. We locate the Article to delete from the database, and then we call the method `#destroy` on that particular instance. This is the proper way to delete a record from the database and we will explain more about this `ActiveRecord` statement in a later chapter that will deal with `ActiveRecord` API.

`destroy` View Template

Having implemented the `destroy` action, let's now implement the corresponding view template. This is the content of the file `app/views/articles/destroy.html.erb` that you need to implement:

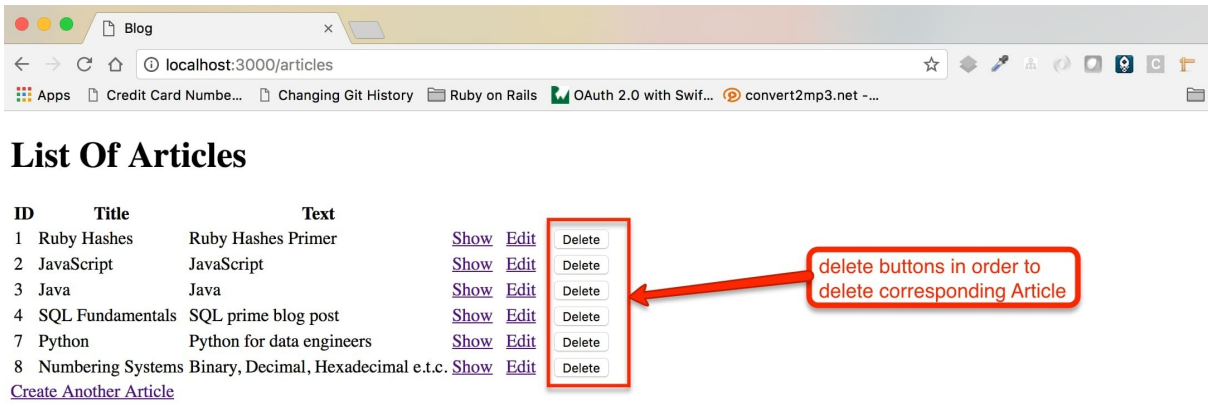
```
<% # File: app/views/articles/destroy.html.erb %>

<h1>Article Has Been Deleted</h1>

<div>
  <a href="/articles">List of Articles</a>
</div>

<div>
  <a href="/articles/new">Create a New Article</a>
</div>
```

(the above code snippet online)

This is very simple. It only has the confirmation message that the Article has been deleted, and two links to ease your navigation in the application.

UI to Delete Article

We are almost ready to close the loop of the functionality to delete the Article, but we have not given user the way to actually invoke the deletion. We will do that by adding this functionality next to the <u style="color: blue;">Show</u> and <u style="color: blue;">Edit</u> links in the list of articles page. So, basically, what we will build will be this:



Delete Buttons In List Of Articles Page

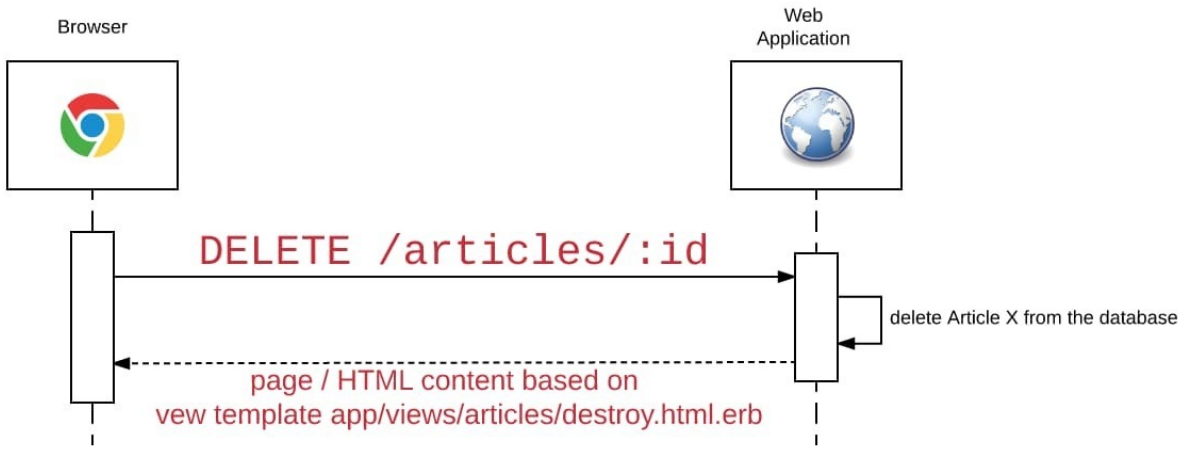
Let's do that. This is the new version of the `app/views/articles/index.html.erb` view template:

```
1  <% # File: app/views/articles/index.html.erb %>
2
3  <h1>List Of Articles</h1>
4
5  <table>
6    <thead>
7      <tr><th>ID</th><th>Title</th><th>Text</th><th></th></tr>
8    </thead>
9    <tbody>
10     <% @articles.each do |article| %>
11       <tr>
12         <td><%= article.id %></td>
13         <td><%= article.title %></td>
14         <td><%= article.text %></td>
15         <td>
16           <a href="/articles/<%= article.id%>">Show</a>  
17           <a href="/articles/<%= article.id%>/edit">Edit</a>   
18           <form method="post" action="/articles/<%= article.id %>" style="display\
19 : inline;">
20             <input type="hidden" name="_method" value="delete"/>
21             <input type="hidden" name="authenticity_token" value="<%= form_authen\
22 ticity_token %>"/>
23             <button type="submit">Delete</button>
24           </form>
25         </td>
26       </tr>
27     <% end %>
28   </tbody>
29 </table>
30 <div>
31   <a href="/articles/new">Create Another Article</a>
32 </div>
```

(the above code snippet online)

The new lines here are lines 18 to 22. For each row, i.e. for each Article in the table with Articles, we have added a form that will issue a `POST` request to `/articles/:id`. However, we use the trick of the hidden field with name `_method` and value `delete` to tell Rails to handle it as a `DELETE` request. Also, dont' forget that we need to make sure we include a hidden field with an authenticity token, which makes sure that our page will not have any security issues. The button `Delete` will post the data to the server.

Perfect! We have now closed the loop of the delete functionality too:



Browser Sends To Delete Endpoint And Gets Back What Destroy View Template Has

The following video demonstrates the delete feature:

Blog Demo Articles Management First Part

Integrate Twitter Bootstrap

We have gone through all the basic components of Rails by building these two sample applications. The `hello_rails` and the `blog` with Articles application. Before, we continue, let's see how we can make this blog application look a little bit nicer. We will integrate it with Twitter Bootstrap.

Twitter Bootstrap Integration Gem

The easiest way that you can take to integrate Twitter Bootstrap with Rails is via the gem bootstrap-sass.

1. Open your Gemfile and add the following two gems into your gem dependencies:

```
gem 'jquery-rails'
gem 'bootstrap-sass'
```

preferably before `gem 'sass-rails'` line. 1. Run `bundle`. 1. Rename the file `app/assets/stylesheets/applicatio` to `app/assets/stylesheets/application.scss`. In other words, you turn this file from a CSS file to a SCSS file that can now include SASS commands. 1. Update the content of the `app/assets/stylesheets/application.scss` file to be as follows:

```
1. // File: app/assets/stylesheets/application.scss
2. //
3. @import "bootstrap-sprockets";
4. @import "bootstrap";
```

1. Open file `app/assets/javascripts/application.js` and add the following lines:

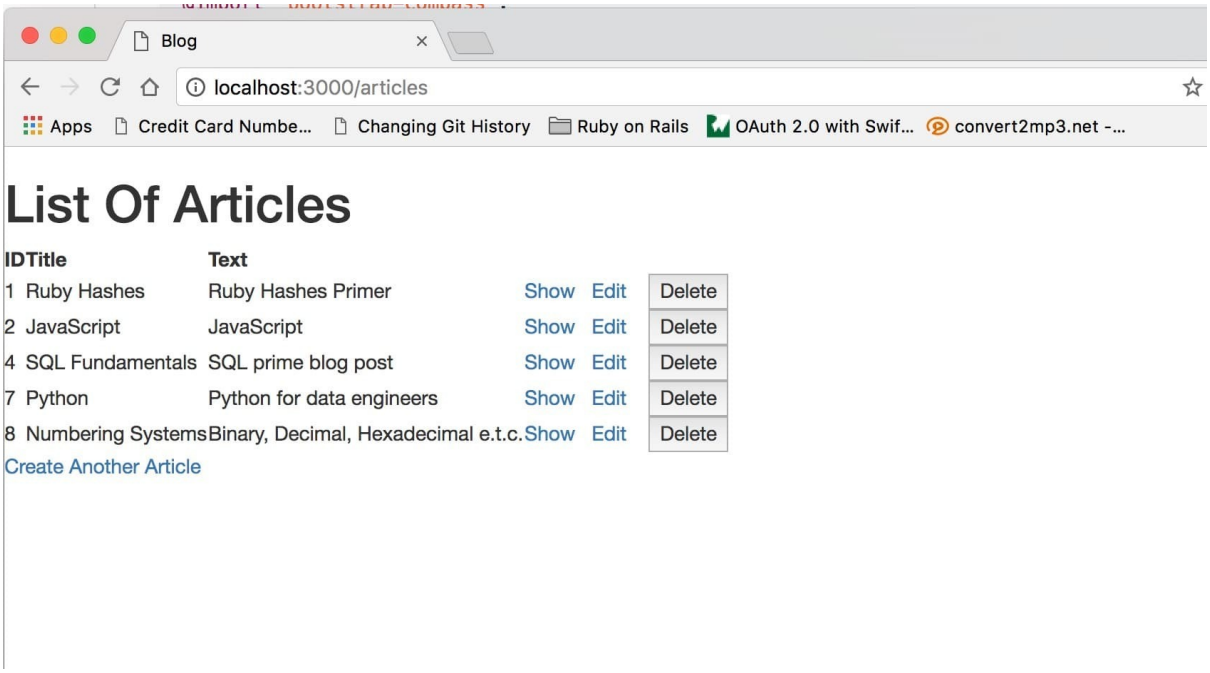
```
//= require jquery
//= require bootstrap-sprockets
```

so that your final `application.js` file needs to be something like this:

```
//= require jquery
//= require bootstrap-sprockets
//= require rails-ujs
//= require turbolinks
//= require_tree .
```

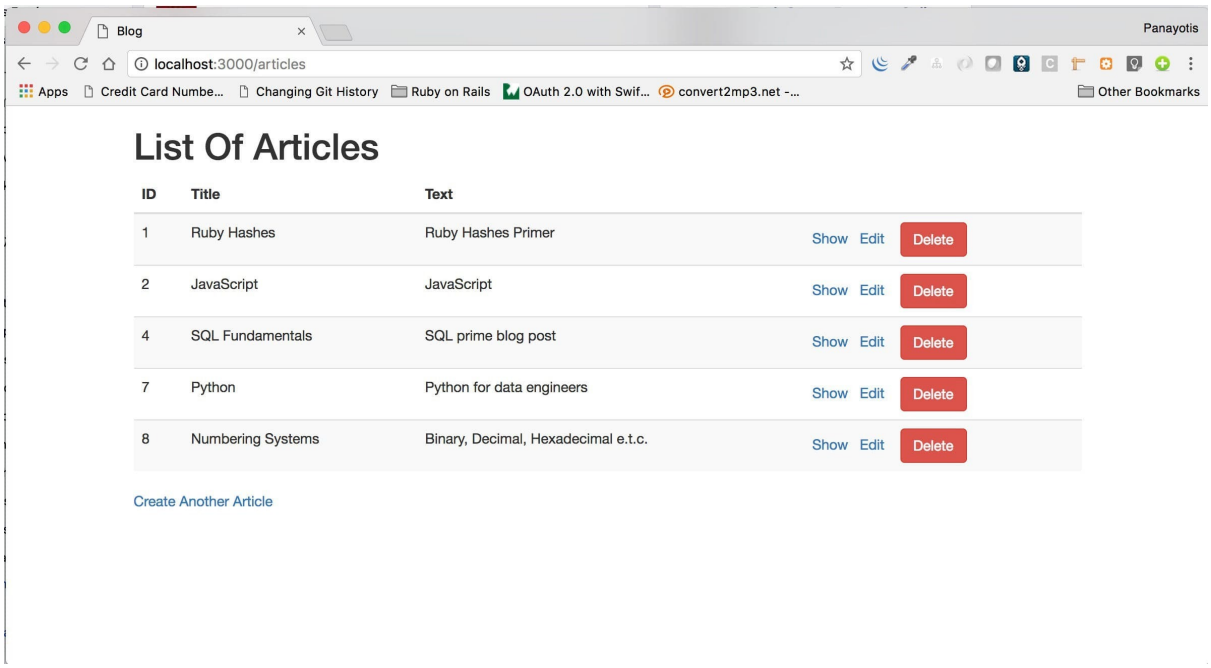
1. Restart your server, with `bin/rails server`. This will take into account the changes you did above.

Now that you have restarted you server, visit the page http://localhost:3000/articles. Do you see any difference in style? It is the Twitter Bootstrap default style applied.



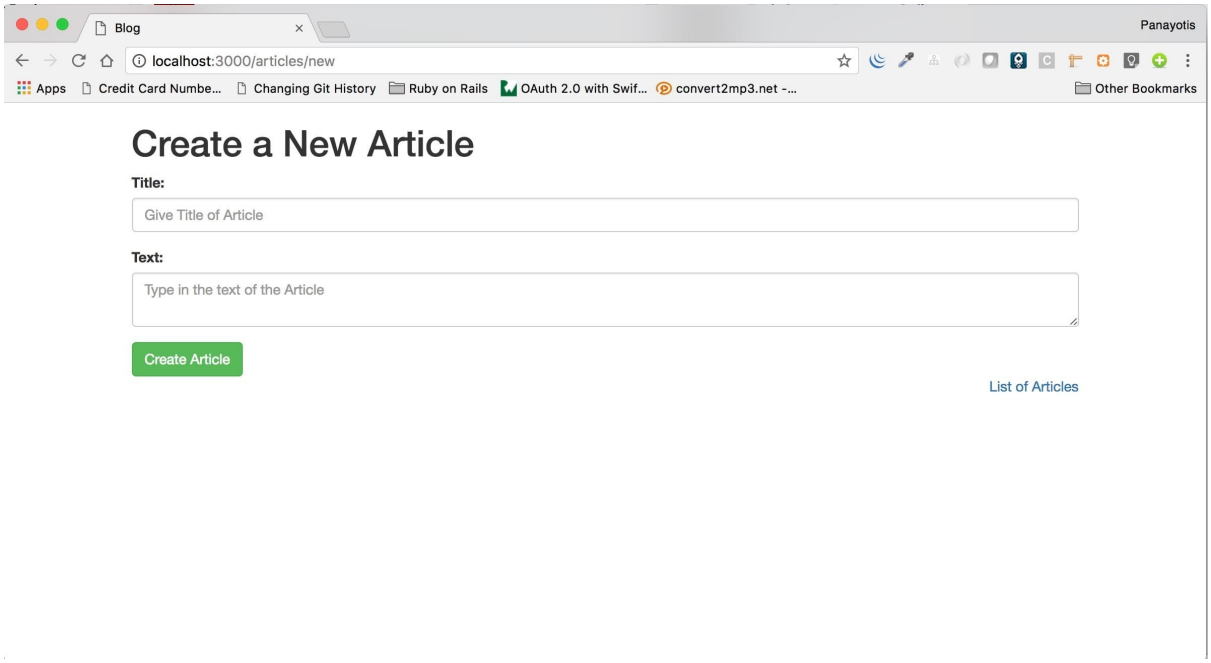
List of Articles With Twitter Bootstrap Applied - Default Styling

Now, go ahead and apply your Twitter Bootstrap knowledge to make this project, the `blog`, looking nice and good. For example, this is how my list of articles takes advantage of Twitter Bootstrap tables:



List of Articles With Twitter Bootstrap Tables

And this is the form to create a new Article, enhanced with the Twitter Bootstrap form related classes:



Twitter Bootstrap Form Styling for Creating an Article

Reminder for Github

Don't forget to commit all your work and push it to your remote Github repository.

Deploy to Heroku

So far so good. We have our first Rails application that can run locally and it is integrated with Twitter Bootstrap. How can we deploy that to a public Web server? Here is where Heroku comes into the picture.

Let's see how we can install our blog into Heroku.

Heroku Login

Run the command `heroku login` and give your heroku credentials. This will allow your local machine to have a login status to your Heroku account.

Add PostgreSQL Support

Heroku does not support SQLite database. It supports PostgreSQL and for that reason, we will need to install the corresponding gem so that we can have PostgreSQL support in our application.

Hence, go ahead and add the following lines to your `Gemfile`:

```
gem 'pg', group: :production
```

(the above code snippet online)

Also, alter the `Gemfile` that refers to `sqlite3` to be included only in `development` and `test` environments:

```
gem 'sqlite3', group: [:development, :test]
```

(the above code snippet online)

Then, run `bundle` to bring in the `pg` (postgres gem) in. We have told Rails that the gem `pg`, for PostgreSQL support, will only be used for the `production` environment and not for development. This is necessary for Heroku deployment.

> **Important**: Our recommendation is that you install PostgreSQL on your local machine too and do your development against PostgreSQL too. So that your development and production environments match as much as possible. In the next chapter we will show you how you can remove the integration with SQLite and PostgreSQL for your local machine too.

Next, you will need to edit your `config/database.yml` file to have the correct settings for the `production` environment:

```
1  # SQLite version 3.x
2  #   gem install sqlite3
3  #
4  #   Ensure the SQLite 3 gem is defined in your Gemfile
5  #   gem 'sqlite3'
6  #
7  default: &default
8    adapter: sqlite3
9    pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
10   timeout: 5000
11
12 development:
13   <<: *default
14   database: db/development.sqlite3
15
16 # Warning: The database defined as "test" will be erased and
17 # re-generated from your development database when you run "rake".
18 # Do not set this db to the same as development or production.
19 test:
20   <<: *default
21   database: db/test.sqlite3
22
23 production:
24   <<: *default
25   adapter: postgresql
26   encoding: unicode
27   database: blog_production
28   username: blog
29   password: <%= ENV['BLOG_DATABASE_PASSWORD'] %>
```

(the above code snippet online)

Watch out for the lines that deal with the `production`, lines 23 to 29. These are the necessary settings for your blog application to run on Heroku production environment.

Specify Ruby Version In Your Gemfile

Heroku needs to know which version your application will be using in production. It does that by reading the corresponding statement inside the `Gemfile`. Add the following, at the end of your `Gemfile`:

```
ruby '2.3.3'
```

(the above code snippet online)

Commit All Your Changes

Commit all your changes.

Create Heroku Application

Create your Heroku application. Use a unique name for your application. This is what I did:

```
blog $ heroku create panosm-blog
Creating □ panosm-blog... done
https://panosm-blog.herokuapp.com/ | https://git.heroku.com/panosm-blog.git
blog $
```

(the above code snippet online)

Confirm Remote

Run the following command to confirm that a new remote has been added to your project:

```
blog $ git remote -v
heroku  https://git.heroku.com/panosm-blog.git (fetch)
heroku  https://git.heroku.com/panosm-blog.git (push)
origin  git@github.com:pmatsinopoulos/blog.git (fetch)
origin  git@github.com:pmatsinopoulos/blog.git (push)
blog $
```

(the above code snippet online)

Do you see the `heroku` remote. This is where you will be pushing your code whenever you will be deploying to production. This is the next step:

Deploy Your Application

Deploy your application. Heroku is heavily integrated with `git` and this is how you deploy your application to Heroku:

```
blog $ git push heroku master
Counting objects: 133, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (116/116), done.
Writing objects: 100% (133/133), 26.75 KiB | 0 bytes/s, done.
Total 133 (delta 24), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote:  !     Warning: Multiple default buildpacks reported the ability to handl\
e this app. The first buildpack in the list below will be used.
remote:                        Detected buildpacks: Ruby,Node.js
remote:                        See https://devcenter.heroku.com/articles/buildpa\
cks#buildpack-detect-order
```

```
remote: -----> Ruby app detected
remote: -----> Compiling Ruby/Rails
remote: -----> Using Ruby version: ruby-2.3.3
remote: -----> Installing dependencies using bundler 1.13.7
remote:        Running: bundle install --without development:test --path vendor/b\
undle --binstubs vendor/bundle/bin -j4 --deployment
remote:        Warning: the running version of Bundler (1.13.7) is older than the\
 version that created the lockfile (1.14.6). We suggest you upgrade to the latest\
 version of Bundler by running `gem install bundler`.
remote:        Fetching gem metadata from https://rubygems.org/.........
remote:        Fetching version metadata from https://rubygems.org/..
remote:        Fetching dependency metadata from https://rubygems.org/.
remote:        Installing rake 12.0.0
remote:        Installing i18n 0.8.1
remote:        Installing concurrent-ruby 1.0.5
remote:        Installing minitest 5.10.2
remote:        Installing thread_safe 0.3.6
remote:        Installing builder 3.2.3
remote:        Installing erubi 1.6.0
remote:        Installing mini_portile2 2.1.0
remote:        Installing websocket-extensions 0.1.2
remote:        Installing nio4r 2.0.0 with native extensions
remote:        Installing rack 2.0.2
remote:        Installing mime-types-data 3.2016.0521
remote:        Installing arel 8.0.0
remote:        Installing execjs 2.7.0
remote:        Installing coffee-script-source 1.12.2
remote:        Installing sass 3.4.23
remote:        Installing method_source 0.8.2
remote:        Installing thor 0.19.4
remote:        Installing multi_json 1.12.1
remote:        Installing pg 0.20.0 with native extensions
remote:        Installing puma 3.8.2 with native extensions
remote:        Using bundler 1.13.7
remote:        Installing tilt 2.0.7
remote:        Installing turbolinks-source 5.0.3
remote:        Installing nokogiri 1.7.2 with native extensions
remote:        Installing tzinfo 1.2.3
remote:        Installing websocket-driver 0.6.5 with native extensions
remote:        Installing mime-types 3.1
remote:        Installing rack-test 0.6.3
remote:        Installing sprockets 3.7.1
remote:        Installing autoprefixer-rails 7.1.1
remote:        Installing uglifier 3.2.0
remote:        Installing coffee-script 2.4.1
remote:        Installing turbolinks 5.0.1
```

```
remote:        Installing activesupport 5.1.1
remote:        Installing mail 2.6.5
remote:        Installing bootstrap-sass 3.3.7
remote:        Installing globalid 0.4.0
remote:        Installing activemodel 5.1.1
remote:        Installing jbuilder 2.6.4
remote:        Installing activejob 5.1.1
remote:        Installing activerecord 5.1.1
remote:        Installing rails-dom-testing 2.0.3
remote:        Installing loofah 2.0.3
remote:        Installing rails-html-sanitizer 1.0.3
remote:        Installing actionview 5.1.1
remote:        Installing actionpack 5.1.1
remote:        Installing actionmailer 5.1.1
remote:        Installing actioncable 5.1.1
remote:        Installing railties 5.1.1
remote:        Installing sprockets-rails 3.2.0
remote:        Installing coffee-rails 4.2.1
remote:        Installing rails 5.1.1
remote:        Installing sass-rails 5.0.6
remote:        Installing jquery-rails 4.3.1
remote:        Bundle complete! 19 Gemfile dependencies, 55 gems now installed.
remote:        Gems in the groups development and test were not installed.
remote:        Bundled gems are installed into ./vendor/bundle.
remote:        Bundle completed (23.36s)
remote:        Cleaning up the bundler cache.
remote:        Warning: the running version of Bundler (1.13.7) is older than the\
 version that created the lockfile (1.14.6). We suggest you upgrade to the latest\
 version of Bundler by running `gem install bundler`.
remote: -----> Installing node-v6.10.0-linux-x64
remote: -----> Detecting rake tasks
remote: -----> Preparing app for Rails asset pipeline
remote:        Running: rake assets:precompile
remote:        Yarn executable was not detected in the system.
remote:        Download Yarn at https://yarnpkg.com/en/docs/install
remote:        I, [2017-05-21T17:12:41.325166 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/application-902d5dce6d9085afdce5\
ebe97998082799250ff7a4dc426f1236cf2f13795551.js
remote:        I, [2017-05-21T17:12:41.325793 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/application-902d5dce6d9085afdce5\
ebe97998082799250ff7a4dc426f1236cf2f13795551.js.gz
remote:        I, [2017-05-21T17:12:48.473756 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/application-895ae8ca2b3268d61532\
66e6779a6d89796096e2da27247f1558a72aa570cff9.css
remote:        I, [2017-05-21T17:12:48.474431 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/application-895ae8ca2b3268d61532\
```

```
66e6779a6d89796096e2da27247f1558a72aa570cff9.css.gz
remote:        I, [2017-05-21T17:12:48.474934 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-13634da87d9e23f8c3ed9108ce1724d183a39ad072e73e1b3d8cbf646d2d0407.eot
remote:        I, [2017-05-21T17:12:48.475451 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-13634da87d9e23f8c3ed9108ce1724d183a39ad072e73e1b3d8cbf646d2d0407.eot.gz
remote:        I, [2017-05-21T17:12:48.476327 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-fe185d11a49676890d47bb783312a0cda5a44c4039214094e7957b4c040ef11c.woff2
remote:        I, [2017-05-21T17:12:48.477181 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-a26394f7ede100ca118eff2eda08596275a9839b959c226e15439557a5a80742.woff
remote:        I, [2017-05-21T17:12:48.477940 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-e395044093757d82afcb138957d06a1ea9361bdcf0b442d06a18a8051af57456.ttf
remote:        I, [2017-05-21T17:12:48.478416 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-e395044093757d82afcb138957d06a1ea9361bdcf0b442d06a18a8051af57456.ttf.gz
remote:        I, [2017-05-21T17:12:48.479191 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-42f60659d265c1a3c30f9fa42abcbb56bd4a53af4d83d316d6dd7a36903c43e5.jpg
remote:        I, [2017-05-21T17:12:48.479659 #1262]  INFO -- : Writing /tmp/buil\
d_00ec1f5ac143f2da5c3d6a880f37a9af/public/assets/bootstrap/glyphicons-halflings-r\
egular-42f60659d265c1a3c30f9fa42abcbb56bd4a53af4d83d316d6dd7a36903c43e5.jpg.gz
remote:        Asset precompilation completed (13.25s)
remote:        Cleaning assets
remote:        Running: rake assets:clean
remote:
remote: ###### WARNING:
remote:        No Procfile detected, using the default web server.
remote:        We recommend explicitly declaring how to boot your server process \
via a Procfile.
remote:        https://devcenter.heroku.com/articles/ruby-default-web-server
remote:
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> console, rake, web, worker
remote:
remote: -----> Compressing...
remote:        Done: 36.4M
remote: -----> Launching...
remote:        Released v5
remote:        https://panosm-blog.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
```

```
To https://git.heroku.com/panosm-blog.git
 * [new branch]      master -> master
blog $
```

(the above code snippet online)

The above is the output of the deployment, but your deployment is missing the creation and initialization of the database.

Initialize Database

Run the following command to initialize your database on production:

```
blog $ heroku run rake db:migrate --app panosm-blog
Running rake db:migrate on ⬢ panosm-blog... up, run.2256 (Free)
D, [2017-05-21T17:15:31.313580 #4] DEBUG -- :    (14.4ms)  CREATE TABLE "schema_m\
igrations" ("version" character varying NOT NULL PRIMARY KEY)
D, [2017-05-21T17:15:31.329135 #4] DEBUG -- :    (11.7ms)  CREATE TABLE "ar_inter\
nal_metadata" ("key" character varying NOT NULL PRIMARY KEY, "value" character va\
rying, "created_at" timestamp NOT NULL, "updated_at" timestamp NOT NULL)
D, [2017-05-21T17:15:31.331039 #4] DEBUG -- :    (0.8ms)  SELECT pg_try_advisory_\
lock(7544364894950660595);
D, [2017-05-21T17:15:31.348311 #4] DEBUG -- :    (1.0ms)  SELECT "schema_migratio\
ns"."version" FROM "schema_migrations" ORDER BY "schema_migrations"."version" ASC
I, [2017-05-21T17:15:31.349801 #4]  INFO -- : Migrating to CreateArticles (201705\
19062431)
D, [2017-05-21T17:15:31.352642 #4] DEBUG -- :    (0.6ms)  BEGIN
== 20170519062431 CreateArticles: migrating ===================================
-- create_table(:articles)
D, [2017-05-21T17:15:31.361801 #4] DEBUG -- :    (7.4ms)  CREATE TABLE "articles"\
 ("id" bigserial primary key, "title" character varying, "text" text, "created_at\
" timestamp NOT NULL, "updated_at" timestamp NOT NULL)
   -> 0.0090s
== 20170519062431 CreateArticles: migrated (0.0091s) ==========================

D, [2017-05-21T17:15:31.367788 #4] DEBUG -- :   SQL (1.0ms)  INSERT INTO "schema_\
migrations" ("version") VALUES ($1) RETURNING "version"  [["version", "2017051906\
2431"]]
D, [2017-05-21T17:15:31.371033 #4] DEBUG -- :    (2.9ms)  COMMIT
D, [2017-05-21T17:15:31.377678 #4] DEBUG -- :   ActiveRecord::InternalMetadata Lo\
ad (0.8ms)  SELECT  "ar_internal_metadata".* FROM "ar_internal_metadata" WHERE "a\
r_internal_metadata"."key" = $1 LIMIT $2  [["key", "environment"], ["LIMIT", 1]]
D, [2017-05-21T17:15:31.386638 #4] DEBUG -- :    (0.5ms)  BEGIN
D, [2017-05-21T17:15:31.388712 #4] DEBUG -- :   SQL (0.8ms)  INSERT INTO "ar_inte\
rnal_metadata" ("key", "value", "created_at", "updated_at") VALUES ($1, $2, $3, $\
4) RETURNING "key"  [["key", "environment"], ["value", "production"], ["created_a\
t", "2017-05-21 17:15:31.386938"], ["updated_at", "2017-05-21 17:15:31.386938"]]
```

```
D, [2017-05-21T17:15:31.390494 #4] DEBUG -- :    (1.4ms)  COMMIT
D, [2017-05-21T17:15:31.391476 #4] DEBUG -- :    (0.7ms)  SELECT pg_advisory_unlo\
ck(7544364894950660595)
blog $
```

(the above code snippet online)

The above confirms that you have initialized the database successfully.

Restart Your Web Server

Now restart your Web server on production so that it takes the changes in the database:

```
blog $ heroku ps:restart --app panosm-blog
Restarting dynos on □ panosm-blog... done
```

(the above code snippet online)

Visit Web App and Create Articles

Now visit the Web application URL and create some articles. Note that your production database does not include the articles that you have created locally.

Note that opening the root page of your Web application it fails, because we have not created a welcome page (like we did for `hello_rails` application). The creation of the welcome page is going to be part of your task for this chapter. But, until then, you can visit the `/articles` path to see the list of Articles and click on create to create new Articles.

This is how my production application looks like:

Blog Demo Articles Management First Part

Closing Note

That was a walk through the main components of Rails and how you can develop a very basic application with Rails. In the next chapter we will polish it and make it be more *Railsish*, i.e. to follow better practices for Rails.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

Important About the apps developed during the chapter content study:

1. If you haven't done so, you need to upload the `hello_rails` and `blog` application code to your Github account.
2. If you haven't done so, you need to publish the `blog` application to your Heroku account and create a public Web application for demo purposes.

Task A

1. Create a welcome page for the `blog` application, like you did for the `hello_rails` application.
2. Make sure that you commit your work and push it to remote repository.
3. Also push your work to your Heroku production server.

Task B

This is going to be the first part of a larger Rails project that you will build while taking the Ruby on Rails section chapters. It will be an e-shop.

In this chapter we start with the main resource, i.e. the `Product`. You will have to build a Rails application that will allow the user to CRUD Products.

The Product needs to have the following properties:

1. code. A string.
2. display_name. A string.
3. display_description. A text.

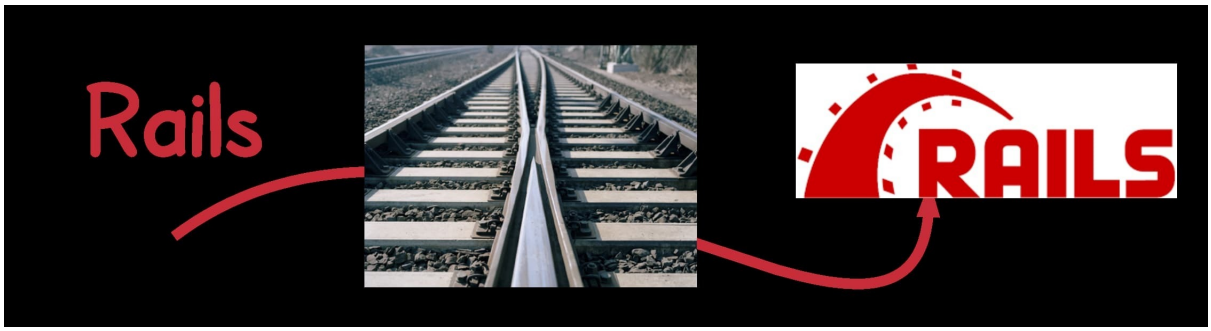
Make sure that you also have a welcome page.

You need to integrate your e-shop with Twitter Bootstrap.

Don't forget to push your code to a new Github repository and to a new Heroku project in production.

2 - The Rails Way

Summary



The Rails Way - Part 1

In the previous chapter we dealt with the basics when doing Rails development. In this chapter we take advantage of the Rails API to write less code with more functionality. We are going to replace parts of the code that we have written in the previous chapter, with code that makes our Web application follow the Rails conventions and take advantage of its rich API.

Learning Goals

1. Learn how to install PostgreSQL on your local machine for development.
2. Learn how to use the `psql` command line interface.
3. Learn how to configure your Web application to use your local PostgreSQL server while in development.
4. Learn how to avoid committing sensitive information, like a password, into your public version control repo.
5. Learn how to avoid polluting the shell environment variable space.
6. Learn how routes code can be simplified with the use of *resources*.
7. Learn about the form builder helper.
8. Learn about form input helpers.
9. Learn how to adapt your controller code to comply with the code that is generated with form tag helpers.
10. Learn about path and URL helpers.
11. Learn about how to parse the parameters posted by a form.
12. Learn how to add validations to your Models.
13. Learn how to handle errors at the controller and view template level.
14. Learn how to preserve the values that user enters on a form.
15. Learn how to inform user that something was wrong with the data they have submitted.
16. Learn about redirections.
17. Learn how you can include your own style files.
18. Learn how to include Twitter Bootstrap alerts.
19. Learn about the flash Hash.
20. Learn how to print the flash content inside a view template.

21. Learn how to take advantage of layout to display information common to all views.
22. Learn how to generate anchors with corresponding view helpers.
23. Learn how to create delete links.
24. Learn how to attach confirmations on the delete links.
25. Learn how to add data HTML attributes.
26. Learn how to reduce code duplication in your controllers using hooks.
27. Learn how to reduce duplication at the view templates level, with the help of partials.
28. Learn about the Rails console.
29. Learn about HAML and its basic rules.

Introduction

Let's continue our `blog` application development. We will work on things that need to be implemented in a Rails-suggested way and using the Rails API.

PostgreSQL Database on Local Machine

Before going into the application code itself, we would like to take some time to make sure that we have the least possible impendance mismatch between our local development environment VS the production environment.

One thing that we need to make sure is the same, is the database vendor we use locally. Currently, our `blog` is using SQLite. Whereas our production server, in Heroku, is using PostgreSQL. We are going to make sure that your local environment is using PostgreSQL too. Here is how:

Make sure PostgreSQL is installed and running

On my machine I did `ps -ef | grep 'postgres'` and I saw this:

```
blog $ ps -ef | grep 'postgre'
 501 82998     1   0  2:52AM ??         0:00.08 /Applications/Postgres.app/Conten\
ts/Versions/9.4/bin/postgres -D /Users/panayotismatsinopoulos/Library/Application\
 Support/Postgres/var-9.4
  501 83002 82998   0  2:52AM ??         0:00.00 postgres: checkpointer process  \

  501 83003 82998   0  2:52AM ??         0:00.08 postgres: writer process
  501 83004 82998   0  2:52AM ??         0:00.04 postgres: wal writer process
  501 83005 82998   0  2:52AM ??         0:00.04 postgres: autovacuum launcher pr\
ocess
  501 83006 82998   0  2:52AM ??         0:00.09 postgres: stats collector proces\
s
  501 85430 83968   0  8:36AM ttys000    0:00.00 grep postgre
blog $
```

(the above code snippet online)

This proves that PostgreSQL is running. Also, if I do `psql` on the command line, I am getting into the PostgresSQL command line interface.

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=#
```

(the above code snippet online)

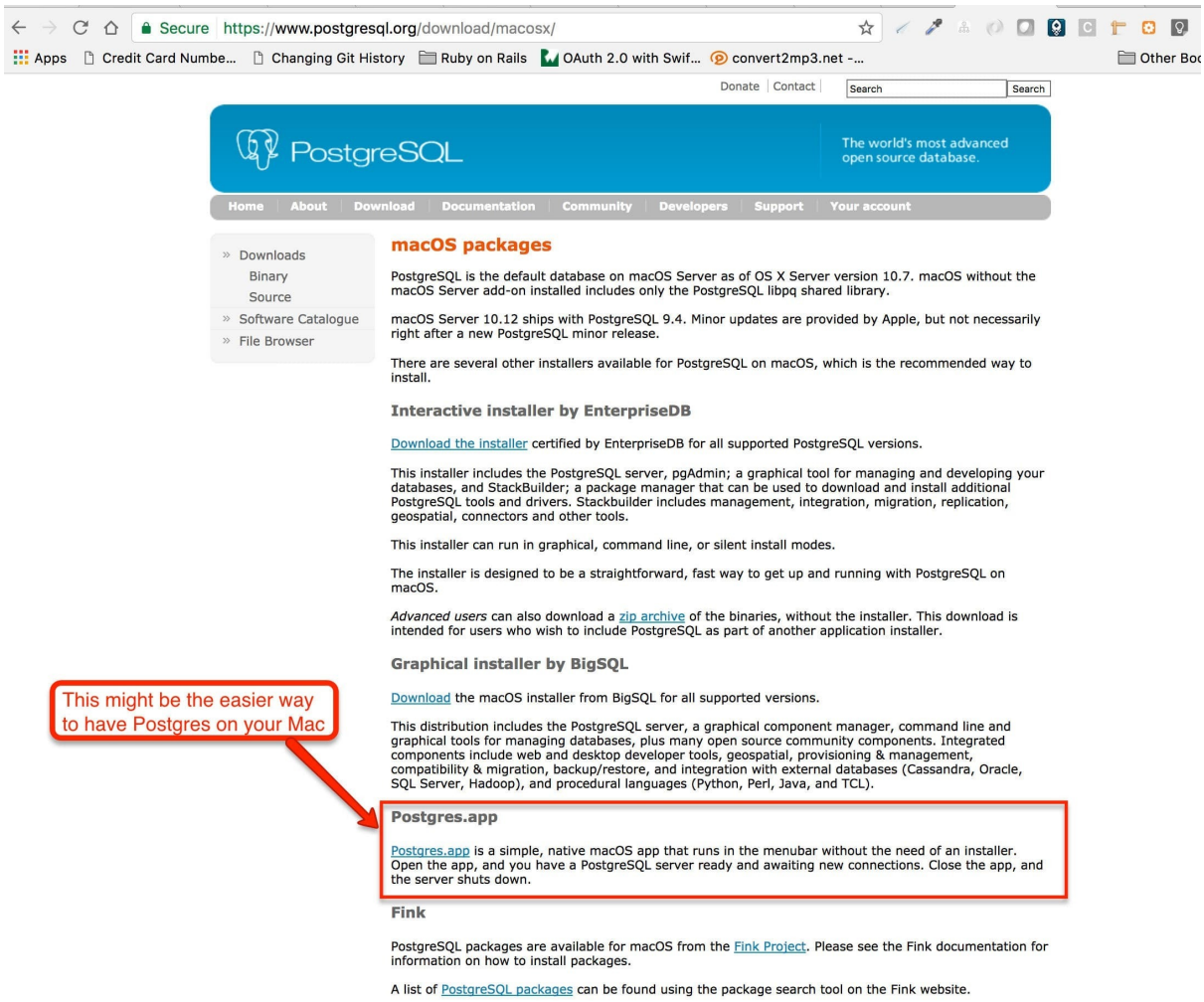
> *Hint:* You can quit the `psql` terminal by giving the command `\q`. Also make sure that, while in `psql` command line, you give the command `help` if you want to learn more about using the `psql` interface.

If you don't have PostgreSQL installed, there are some ways you can install it.

Mac OS X

On my Mac, I have installed the `Postgres.app`, which I found very easy to do and covered my development needs.

On the official download page here you can see also other ways you can install PostgreSQL on your Mac.

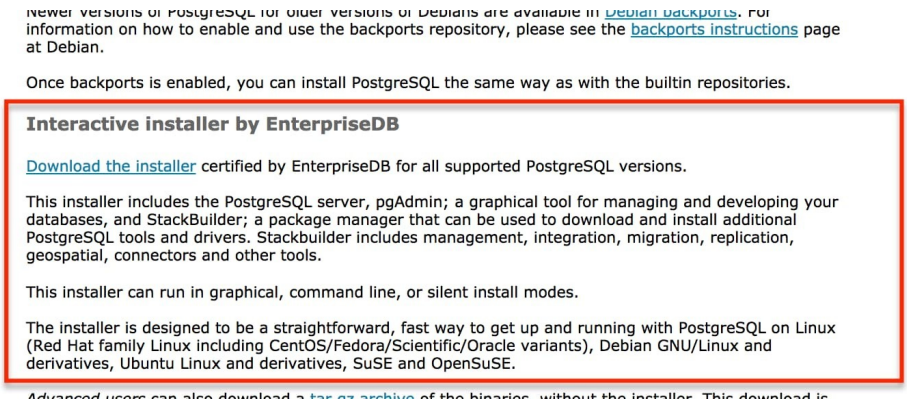


Easy Way to Install PostgreSQL

Linux (Ubuntu/Debian)

- Here is the official page for installing PostgreSQL on Ubuntu.
- Here is the official page for installing PostgreSQL on Debian.

You may want to try the *Interactive Installer by EnteripriseDB*


Newer versions of PostgreSQL for older versions of Debians are available in Debian backports. For information on how to enable and use the backports repository, please see the backports instructions page at Debian.

Once backports is enabled, you can install PostgreSQL the same way as with the builtin repositories.

Interactive installer by EnterpriseDB

Download the installer certified by EnterpriseDB for all supported PostgreSQL versions.

This installer includes the PostgreSQL server, pgAdmin; a graphical tool for managing and developing your databases, and StackBuilder; a package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder includes management, integration, migration, replication, geospatial, connectors and other tools.

This installer can run in graphical, command line, or silent install modes.

The installer is designed to be a straightforward, fast way to get up and running with PostgreSQL on Linux (Red Hat family Linux including CentOS/Fedora/Scientific/Oracle variants), Debian GNU/Linux and derivatives, Ubuntu Linux and derivatives, SuSE and OpenSuSE.

Interactive Installer by EnterpriseDB

After Installation

Make sure that you can access the `psql` command line interface and that you can start and stop your PostgreSQL server.

Update `config/database.yml` file

Now that you have PostgreSQL up and running on your local machine, you will need to update your `config/database.yml` file to have the correct settings for the `development` and `test` environments. Currently, they are using SQLite, but you want them to be using PostgreSQL.

This is the new version of the `config/database.yml` file:

```
# File: config/database.yml
#
default: &default
  adapter: postgresql
  encoding: unicode

development:
  <<: *default
  database: blog_development

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  <<: *default
  database: blog_test

production:
  <<: *default
  database: blog_production
```

(the above code snippet online)

As you can see, we have an area with the default settings. Then we have environment specific settings. For `development`, `test` and `production`. Note that the environment specific settings inherit from the `default` settings.

> **Important**: We are not using `username` and `password`, but you may want to use them in your setup. We are not using because we have setup our PostgreSQL security checks on our local machine to allow us for connection without credentials. If you have not done the same, you may need to add the `username` and `password` keys in your `development` and `test` blocks. However, don't write the password verbatim into the `config/database.yml` file, because this will risk you committing that to your git repo and pushing it to public Github remote repository. Use the ERB tags to tell `config/database.yml` to take the `password` from an environment variable. This is an example `config/database.yml` that does that:

```
# File: config/database.yml
#
default: &default
  adapter: postgresql
  encoding: unicode

development:
  <<: *default
  database: blog_development
  username: blog
  password: <%= ENV['BLOG_PASSWORD_DEVELOPMENT']

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  <<: *default
  database: blog_test
  username: blog
  password: <%= ENV['BLOG_PASSWORD_TEST']

production:
  <<: *default
  database: blog_production
```

> Having done the above, you will need to set the environment variables in your shell so that they hold the actual password values. See later on in this chapter how we use a gem to help us set the environment variables while we are doing development and test.
>
> Finally, the `username` and `password` do not play any role for the `production` block, because Heroku ignores them anyway and sets up the connection to the database with its own credentials.

Hint: If you use username and password to start `psql` on your shell prompt, then you will have to use these in your `development` and `test` blocks in your `config/database.yml`. If you are not required to give these credentials and you have free access to `psql`, then you will not need to use `username` and `password`.

Update `Gemfile`

Now, the `Gemfile` defines a dependency to `sqlite3` for `development` and `test` environments. This line should be removed. Also, the line that defines a dependency to `pg` gem only for `production` environment, it needs to be updated by removing the environment predicate. This is how your `Gemfile` needs to be:

```
source 'https://rubygems.org'

git_source(:github) do |repo_name|
  repo_name = "#{repo_name}/#{repo_name}" unless repo_name.include?("/")
  "https://github.com/#{repo_name}.git"
end


# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.1.1'
# Use Puma as the app server
gem 'puma', '~> 3.7'
gem 'jquery-rails'
gem 'bootstrap-sass'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# See https://github.com/rails/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.2'
# Turbolinks makes navigating your web application faster. Read more: https://git\
hub.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.5'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 3.0'
# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Capistrano for deployment
```

```
# gem 'capistrano-rails', group: :development

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger conso\
le
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
  # Adds support for Capybara system testing and selenium driver
  gem 'capybara', '~> 2.13'
  gem 'selenium-webdriver'
end

group :development do
  # Access an IRB console on exception pages or by using <%= console %> anywhere \
in the code.
  gem 'web-console', '>= 3.3.0'
  gem 'listen', '>= 3.0.5', '< 3.2'
  # Spring speeds up development by keeping your application running in the backg\
round. Read more: https://github.com/rails/spring
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]

gem 'pg'

ruby '2.3.3'
```

(the above code snippet online)

Create Local Databases

Having done that, it is now time for us to create the local database. Previously, we had it with SQLite, but now we need to recreate it using PostgreSQL.

```
blog $ bin/rake db:create
Running via Spring preloader in process 88054
Created database 'blog_development'
Created database 'blog_test'
blog $
```

(the above code snippet online)

Nice. The last two lines of the above output confirm that we have created the database with PostgreSQL.

Schema Migrations

With the database in place, let's run the schema migrations. You will see something like the following:

```
blog $ bin/rake db:migrate
Running via Spring preloader in process 88121
== 20170519062431 CreateArticles: migrating ==================================
-- create_table(:articles)
   -> 0.0052s
== 20170519062431 CreateArticles: migrated (0.0053s) ==========================
blog $
```

(the above code snippet online)

The above confirms that we have run the schema migrations successfully. But we can confirm that with the help of `psql` too, i.e. with the help of the PostgreSQL command line interface:

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \l
                                                                          \
         List of databases
             Name              |         Owner          | Encoding |   Collate   \
|    Ctype    |                 Access privileges
-------------------------------+------------------------+----------+-------------\
+-------------+-----------------------------------------------------
 blog_development              | panayotismatsinopoulos | UTF8     | en_US.UTF-8 \
| en_US.UTF-8 |
 blog_test                     | panayotismatsinopoulos | UTF8     | en_US.UTF-8 \
| en_US.UTF-8 |
...
panayotismatsinopoulos=#
```

(the above code snippet online)

The `\l` command returns the list of databases. Then with `\connect blog_development` we can connect to the database:

```
panayotismatsinopoulos=# \connect blog_development
You are now connected to database "blog_development" as user "panayotismatsinopou\
los".
blog_development=#
```

(the above code snippet online)

And the command to list all the tables of the current database is `\dt`:

```
blog_development=# \dt
                      List of relations
 Schema |         Name         | Type  |         Owner
--------+----------------------+-------+-----------------------
 public | ar_internal_metadata | table | panayotismatsinopoulos
 public | articles             | table | panayotismatsinopoulos
 public | schema_migrations    | table | panayotismatsinopoulos
(3 rows)

blog_development=#
```

(the above code snippet online)

You can see that there 3 tables in our newly created database. The `ar_internal_metadata` and `schema_migrations` are tables that Rails creates and we will talk about those in the later chapters. Our busines domain table is the table `articles`. And you can inspect its structure with the `\d+ articles` command:

```
blog_development=# \d+ articles
                                                        Table "public.articles"
   Column   |            Type             |                      Modifiers      \
                 | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
----------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('articles_id\
_seq'::regclass) | plain    |              |
 title      | character varying           |                                     \
                 | extended |              |
 text       | text                        |                                     \
                 | extended |              |
 created_at | timestamp without time zone | not null                            \
                 | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                 | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)

blog_development=#
```

(the above code snippet online)

Cool. Let's exit the `psql` prompt with the command `\q`.

We are now happy that the integration of our Rails application to our local PostgreSQL server has been done correctly.

Environment Variables

As we said earlier, your development might rely on the values of various Environment Variables. And we already know that Ruby and Rails have access to environment variables via the `Hash ENV`. On the other hand, while doing development, we might not want to pollute the real Environment Variables namespace. We need a way to create *fake* environment variables, those that are only needed by our Rails application while doing development or running tests.

We can do that with the help of the gem dotenv-rails. We will need this gem only for the `development` and `testing` environments. Not for `production`. In `production` the environment variables are set by the hosting shell. We will see this is done for Heroku-deployed Rails appplications.

Let's amend our `Gemfile` to have this gem in.

```
source 'https://rubygems.org'

git_source(:github) do |repo_name|
  repo_name = "#{repo_name}/#{repo_name}" unless repo_name.include?("/")
  "https://github.com/#{repo_name}.git"
end


# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.1.1'
# Use Puma as the app server
gem 'puma', '~> 3.7'
gem 'jquery-rails'
gem 'bootstrap-sass'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# See https://github.com/rails/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.2'
# Turbolinks makes navigating your web application faster. Read more: https://git\
hub.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.5'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 3.0'
# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'
```

```
33
34 # Use Capistrano for deployment
35 # gem 'capistrano-rails', group: :development
36
37 group :development, :test do
38   # Call 'byebug' anywhere in the code to stop execution and get a debugger conso\
39 le
40   gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
41   # Adds support for Capybara system testing and selenium driver
42   gem 'capybara', '~> 2.13'
43   gem 'selenium-webdriver'
44   gem 'dotenv-rails'
45 end
46
47 group :development do
48   # Access an IRB console on exception pages or by using <%= console %> anywhere \
49 in the code.
50   gem 'web-console', '>= 3.3.0'
51   gem 'listen', '>= 3.0.5', '< 3.2'
52   # Spring speeds up development by keeping your application running in the backg\
53 round. Read more: https://github.com/rails/spring
54   gem 'spring'
55   gem 'spring-watcher-listen', '~> 2.0.0'
56 end
57
58 # Windows does not include zoneinfo files, so bundle the tzinfo-data gem
59 gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
60
61 gem 'pg'
62
63 ruby '2.3.3'
```

(the above code snippet online)

Note that we have only added line 42, `gem 'dotenv-rails'`. We have not done any other change. Note that we have added this line inside the gems that we need while we are in `development` or `test`.

Let's run `bundle` now to bring this gem (and it's dependencies) in.

Then create the files `.env` and `.env.test` in the root folder of your project. These files can be initially empty. But later on, if you need an environment variable, you will have to put it there. For example, if your database password for `development` needs to be specified with the environment variable `'BLOG_PASSWORD_DEVELOPMENT'`, then you need to have your `.env` file being with the following content:

```
BLOG_PASSWORD_DEVELOPMENT=dadf#A0(
```

(the above code snippet online)

Each line of the `.env` file contains one environment variable, with its name and its value for the `development` environment. While running tests, the environment variables are being read from the `.env.test` file instead.

> **Important**. You should not add/commit the `.env` and `.env.test` files in your Git repository. In order to avoid doing that, you will have to edit the file `.gitignore` and add two lines that would make Git ignore these files: `bash .env .env.test`

Routes with `resources`

Let's now move to another improvement that will make our code to be more Railsish. Look at the `config/routes.rb` file:

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  get 'articles/new'
  get 'articles/:id/edit', to: 'articles#edit'
  get 'articles/:id', to: 'articles#show'
  get 'articles', to: 'articles#index'
  post 'articles', to: 'articles#create'
  put 'articles/:id', to: 'articles#update'
  delete 'articles/:id', to: 'articles#destroy'

  root 'welcome#index'
end
```

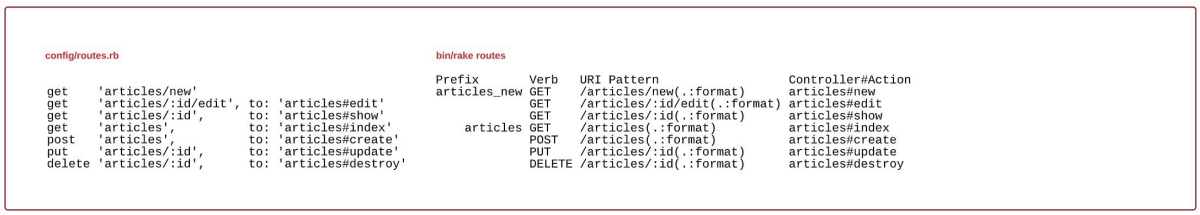
(the above code snippet online)

It defines various routes for the Article resource (besides the route to welcome page). Let's refresh how these entries are being translated to actual routes:

```
blog $ bin/rake routes
      Prefix Verb   URI Pattern                  Controller#Action
articles_new GET    /articles/new(.:format)      articles#new
             GET    /articles/:id/edit(.:format) articles#edit
             GET    /articles/:id(.:format)      articles#show
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
blog $
```

(the above code snippet online)

We have 7 endpoints corresponding to the 7 entries inside the `config/routes.rb` file:



Config Routes Entries Mapped to Actual Routes

We can see how there is one-to-one mapping from the entries in `config/routes.rb` to the actual routes that Rails finally compiles by reading the content of the `config/routes.rb` file.

We have already said that Rails is heavily based on *resources*. For that reason, and because it seems that there is a pattern here with regards to the routes that one usually defines for a resource, Rails provides a quick way to generate all these 7 routes with the use of only one Ruby statement.

Remove the 7 lines from the `config/routes.rb` file that have to do with the Article resource and replace them with the statement `resources :articles`. In other words, your `config/routes.rb` file needs to be like this:

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  resources :articles

  root 'welcome#index'
end
```

(the above code snippet online)

Now, run the command to see which routes Rails finally defines out of this new `config/routes.rb` file:

```
blog $ bin/rake routes
       Prefix Verb   URI Pattern                  Controller#Action
article_index GET    /article(.:format)           article#index
              POST   /article(.:format)           article#create
  new_article GET    /article/new(.:format)       article#new
 edit_article GET    /article/:id/edit(.:format)  article#edit
      article GET    /article/:id(.:format)       article#show
              PATCH  /article/:id(.:format)       article#update
              PUT    /article/:id(.:format)       article#update
              DELETE /article/:id(.:format)       article#destroy
         root GET    /                            welcome#index
```

(the above code snippet online)

Bingo! As you can see, with a single line of code, `resources :articles`, we have generated the same routes that we had before. Actually, there is one more, the `PATCH` verb, but it points to the same controller and action like the one for `PUT` verb.

> *Information:* The `PATCH` verb is preferred when we update parts of a resource. The `PUT` verb is preferred when we update all the attributes of a resource and the request bears the new state of the resource as a whole. That is why the `PATCH` is not idempotent whereas the `PUT` is considered to be. Read more about it here

The differences to the routes generated with the manual entries of Article routes were very slightly different:

bin/rake routes (without resources :articles)

```
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
articles_new GET    /articles/new(.:format)      articles#new
             GET    /articles/:id/edit(.:format) articles#edit
             GET    /articles/:id(.:format)      articles#show

             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
```

bin/rake routes (with resources :articles)

```
      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
```

Differences To The Routes without resources :articles

1. There are some differences in the `Prefix` values. We have not talked about what the prefixes are yet.
2. There is this new `PATCH` entry.

In order to close this `resources` utility, let me just tell you that if you can generate your resources with `resources`, then prefer that. But make sure that you cover all the endpoints. *Covering* all the endpoints might mean more than just implementing the controllers and their actions and views. For example, you may want to deal with security issues (authentication/authorization), something that we will discuss about in later chapters.

Form Builders

When we created the form to create a new Article, we used HTML markup, as we have already learned in the HTML section of this course. However, when you write Rails, the framework provides you some helper methods that make your life much easier, when creating forms. The primary tool is the form builder method `form_for`. Let's now recreate the form to create articles using the `form_for` form builder:

```
<% # File: app/views/articles/new.html.erb %>

<h1>Create a New Article</h1>
<%= form_for :article do |f| %>
  <div class="form-group">
    <%= f.label :title %>
    <%= f.text_field :title, placeholder: 'Give Title of Article', class: 'form-c\
ontrol' %>
  </div>

  <div class="form-group">
    <%= f.label :text %>
    <%= f.text_area :text, placeholder: 'Type in the text of the Article', class:\
 'form-control' %>
  </div>

  <div>
    <%= f.submit class: 'btn btn-success' %>
  </div>
<% end %>

<div class="text-right">
  <a href="/articles">List of Articles</a>
</div>
```

(the above code snippet online)

Here are the things that you should be aware of and that they make this markup differ from the first version we used:

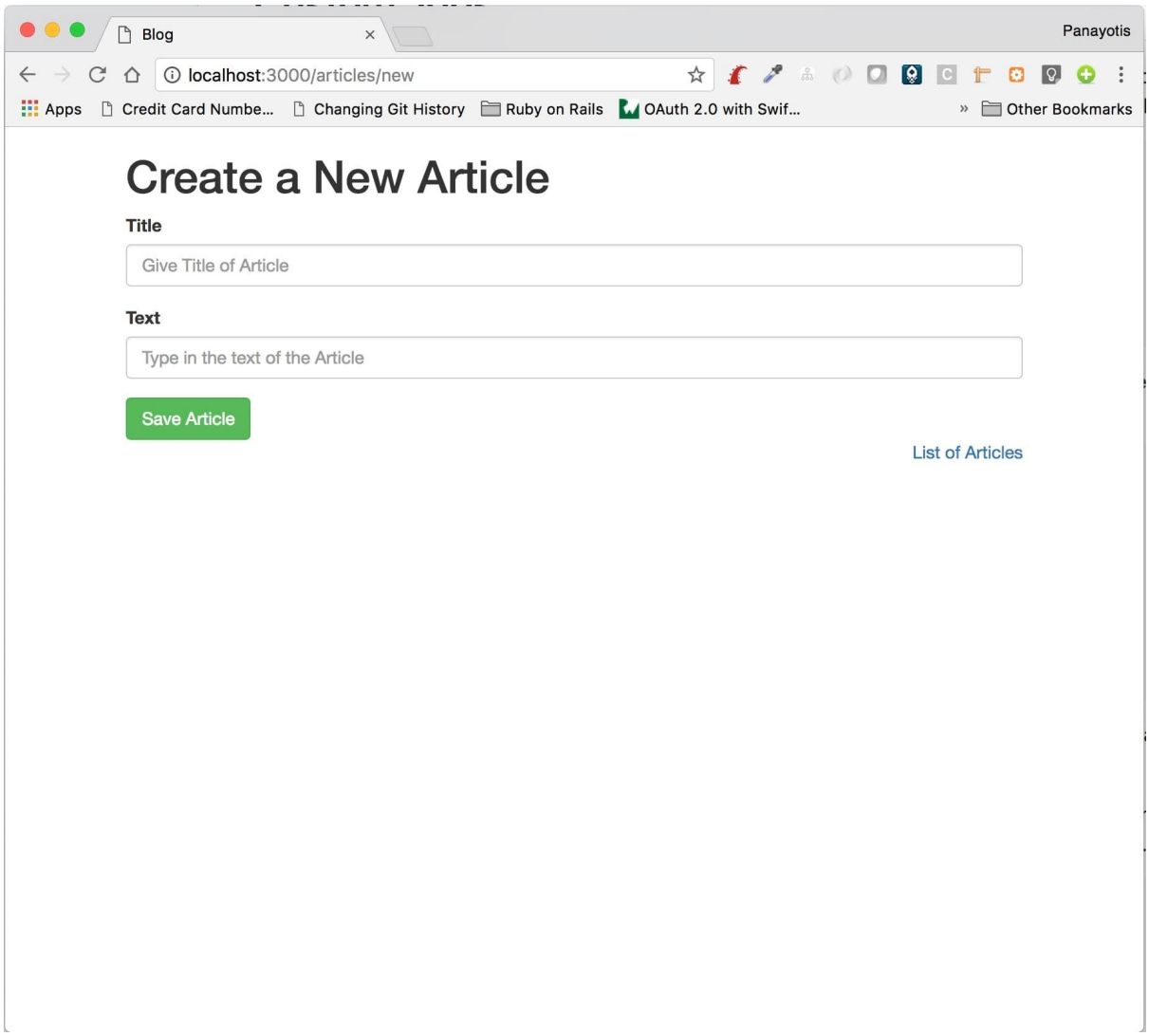
1. We use the `form_for` helper which takes as input the symbol that corresponds to the resource you are building a form for.
2. We provide a block in which we will use both standard HTML and ERB to generate the input related information.

```
<% # File: app/views/articles/new.html.erb %>

<h1>Create a New Article</h1>
<%= form_for :article do |f| %>
 <div class="form-group">
   <%= f.label :title %>
   <%= f.text_field :title, placeholder: 'Give Ti... %>
 </div>

 <div class="form-group">
   <%= f.label :text %>
   <%= f.text_area :text, placeholder: 'Type in ... %>
 </div>

 <div>
   <%= f.submit class: 'btn btn-success' %>
 </div>
<% end %>

<div class="text-right">
 <a href="/articles">List of Articles</a>
</div>
```

do ... end block that will build the form using the form builder.

3. The `f.label` will generate a `label` HTML tag.
4. The `f.text_field` will generate an `input` of type `text` HTML tag.
5. The `f.text_area` will generate a `textarea` HTML tag.
6. The `f.submit` will generate an `input` of type `submit`.

If you save the above `.html.erb` content and visit the URL http://localhost:3000/articles/new, you will see the form to create a new article displayed as such:



Create New Article Generated With form_for

The most important details of this technique (`form_for`) to build the form are revealed if you inspect the source of the generated HTML page. Right-click on the page and select `View Source`. This is the part that is generated for the form:

```
<form action="/articles" accept-charset="UTF-8" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input type="hidden" name="authenticity_token" value="VZVE9vJYsJ2adxLe0TZp1RkUM\
cQwbd313alxaV7OeMPVEVZ+ShGii2X+yn7ypuTTdmYAxZNaxvmhBc4W7UrcEg==" />
  <div class="form-group">
    <label for="article_title">Title</label>
    <input placeholder="Give Title of Article" class="form-control" type="text" n\
ame="article[title]" id="article_title" />
  </div>

  <div class="form-group">
    <label for="article_text">Text</label>
```

```
13      <textarea placeholder="Type in the text of the Article" class="form-control" \
14  name="article[text]" id="article_text">
15      </textarea>
16    </div>
17
18    <div>
19      <input type="submit" name="commit" value="Save Article" class="btn btn-succes\
20  s" data-disable-with="Save Article" />
21    </div>
22  </form>
```

(the above code snippet online)

1. On line 1, do you see the attributes `action` and `method` that have been generated automatically? The `method` has the value `post` as expected. The `action`, does not have the correct value, though. We expected that to be `/articles`, but, instead, it has the value `/articles/new`. POSTing to `/articles/new` is not an endpoint that is covered by our routes. Let's keep that for a while at the back of our head. We will need to fix that later on.
2. Do you see the line 3? It is the `authenticity_token` hidden field that we had to create manually. Now it is created automatically.
3. On line 2, you can see another hidden field that is automatically generated. It is the field with name `utf-8` and value This makes sure that the data of the form are submitted as UTF-8. This is necessary because some browsers do not submit the data correctly.
4. Lines 6 and 11 are the input fields of the form. Their `id` and `name` attributes have been automatically generated. Do you see their convention? The `id` and `name` is constructed within the namespace of the resource. So, instead of `title` now it is `article_title` and `article[title]` respectively.
5. Lines 5 and 10 are the label tags which correctly have the `for` attribute pointing to the corresponding `input` fields. Also, the `label` HTML content is automatically generated too.
6. The `input` field with type `submit` has been automatically created too.

After having done the above analysis, there are some things that we need to do in order to make sure the form creates new articles

`form action` Attribute Value

In order to fix the problem with the `action` value of the `form` generated, we need to give value to the key `url` that we pass to the `form_for` method:

```
<%= form_for :article, url: '/articles' do |f| %>
...
<% end %>
```

(the above code snippet online)

If you do the above change, you refresh the page on your browser and you view the source of the page, you will see that the form now has the correct `action` value:

```
<form action="/articles" accept-charset="UTF-8" method="post">
...
<form>
```

(the above code snippet online)

Change Params Parsing

Now that the `name` attributes of the form have changed, we will need to change the `create` action logic in the `articles_controller`. Here is how the `create` action should be implemented:

```
...
def create
  article = Article.new
  article.title = params[:article][:title]
  article.text = params[:article][:text]
  article.save
end
...
```

(the above code snippet online)

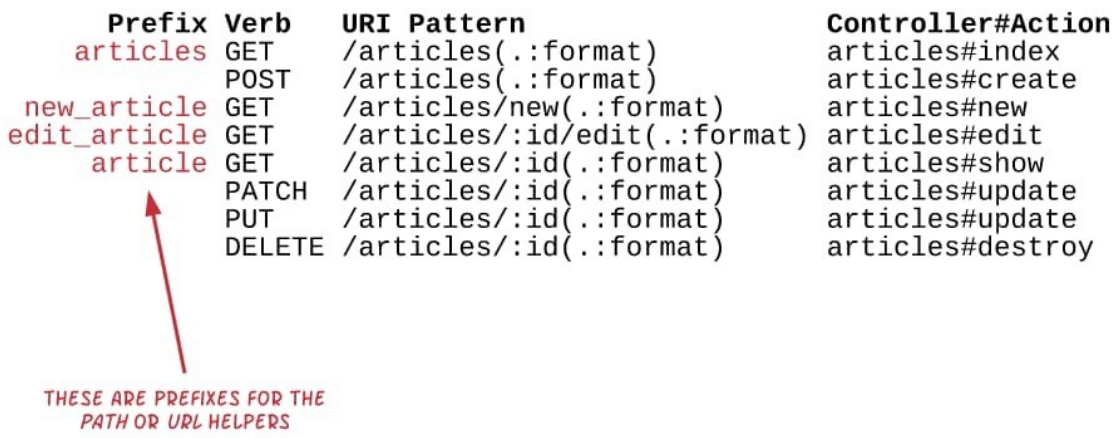
So, instead of `params[:title]`, we will use `params[:article][:title]`. Similarly for the `title` attribute.

Having done all the above, Try to create a new Article. Does it work?

YES! It does.

Path and URL Helpers

We will continue to improve our code for the create new article functionality. We move onto *path and url helpers*. These are methods that we can use in our Views to generate paths and full URLs. You can tell which methods you have available by looking at the *Prefix* column on the `bin/rake routes` output:



Prefixes and Path and URL Helpers

The path helpers are methods whose names are the prefix and the string `_path` appended at the end. The URL helpers are methods whose names are the prefix and the string `_url` appended at the end. So, when `bin/rake routes` generates the routes:

```
blog $ 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
blog $
```

(the above code snippet online)

the available path helpers are:

1. `articles_path`
2. `new_article_path`
3. `edit_article_path`
4. `article_path`
5. `root_path`

and the available URL helpers are:

1. `articles_url`
2. `new_article_url`

3. `edit_article_url`
4. `article_url`
5. `root_url`

Let's now enhance our `app/views/articles/new.html.erb` template to be using path helpers:

```
1  <% # File: app/views/articles/new.html.erb %>
2
3  <h1>Create a New Article</h1>
4  <%= form_for :article, url: articles_path do |f| %>
5    <div class="form-group">
6      <%= f.label :title %>
7      <%= f.text_field :title, placeholder: 'Give Title of Article', class: 'form-c\
8  ontrol' %>
9    </div>
10
11   <div class="form-group">
12     <%= f.label :text %>
13     <%= f.text_area :text, placeholder: 'Type in the text of the Article', class:\
14   'form-control' %>
15   </div>
16
17   <div>
18     <%= f.submit class: 'btn btn-success' %>
19   </div>
20 <% end %>
21
22 <div class="text-right">
23   <a href="<%= articles_path %>">List of Articles</a>
24 </div>
```

(the above code snippet online)

The only amendments are on lines 4 and 21. Instead of using `/articles` we use `articles_path`. Later on we will take advantage of the other helpers too. They are very helpful because they make sure that the paths and URLs are built following any standard and convention and they reduce the possibility to write something wrong, especially when the links might include placeholders and query parameters.

Note that the difference between a `<...>_path` and the corresponding `<...>_url` is that the latter version prepends the protocol, current host and port. In other words, the `articles_url` would have been `http://localhost:3000/articles` instead of just `/articles` that is generated by `articles_path`. Generally, we use the `<...>_path` version, but there are cases that we need to make sure the `<...>_url` version is used. For example, when we generate links that will be embedded inside emails that we send out.

Strong Parameters

We have learned how to access the parameters posted from the form to the server. Here is the `create` action again:

```
def create
  article = Article.new
  article.title = params[:article][:title]
  article.text = params[:article][:text]
  article.save
end
```

(the above code snippet online)

The above implementation implies that one would have to amend it every time a new param is added to the form. There is a cleaner way you can write this.

```
def create
  article = Article.new(article_params)
  article.save
end
```

(the above code snippet online)

The above invokes the `article_params` method which needs to be a `private` method implemented in the `articles_controller.rb`:

```
private

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

(the above code snippet online)

The above implementation uses a Rails API that

1. specifies which namespace we should use to pick up the values of the form. In this case, this is the `:article` one.
2. specifies what keys are allowed/permitted to exist in this bucket/namespace. Only the values of these keys are taken and used as values to create the article.

The above is a very good practice to use. It clearly conveys the design behind the form that you have published and makes sure that no other keys will be processed. Hence, malicious forms will not be able to submit other keys and set values to attributes of an Article that might not be public for creation (or update).

Try to create another Article. Does it work? Yes it does. We will use this new private method for the `update` action too, but later on.

Model Validations

We still stay in the creation of an Article and now we are suggesting you to try to create an Article without giving any of the values, i.e. with neither *title* nor *text*. What will happen when you click to create such an Article? This, with the current implementation, is something that will allow the Article to be created with blank values.

Blog Demo Articles Management First Part

How, can we prevent that? We can definitely prevent that at the client side, but, before we do that, we prefer to make sure that things are shielded from such situations at the server-side.

> **Important**: Delete any Article that you have created and that it does not have either text or title.

Rails provide a mechanism which is called ActiveRecord Validations. Here is what we have to do in order to prevent an Article being created with empty title or empty text:

```ruby
# File: app/models/article.rb
#
class Article < ApplicationRecord
  validates :title, presence: true
  validates :text, presence: true
end
```

(the above code snippet online)

The two lines that we have added are examples of Active Record validations. They both make sure that the corresponding attributes are going to have values before being created or updated.

With the above validations in place, try to create an Article with empty values. Does it work?

You will see that it does not. And this is what you see in the Rails logs:

```
Started POST "/articles" for 127.0.0.1 at 2017-05-28 23:13:44 +0300
Processing by ArticlesController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"mKZEGveBU8ndKZTM5mozaiR9XgS4tf\
nX8ae4DBwIgJkYIlaST8hB3yKgTGzF+r5sSw9vBRuC4tuNCwdzr4wkSA==", "article"=>{"title"=\
>"", "text"=>""}, "commit"=>"Save Article"}
   (0.2ms)  BEGIN
   (0.1ms)  ROLLBACK
  Rendering articles/create.html.erb within layouts/application
  Rendered articles/create.html.erb within layouts/application (0.4ms)
Completed 200 OK in 48ms (Views: 43.3ms | ActiveRecord: 0.3ms)
```

(the above code snippet online)

There is no `INSERT` command at the database level. The insertion didn't take place because the model was considered *invalid*. The validations take place before the actual work to the database.

We will talk more about validations in a later chapter.

The following video demonstrates the fact that the article is not created if information is missing:

Article Is Not Created If Information Is Missing

Handle Error

We have fixed the problem with *invalid* Articles and now `create` action cannot save such erroneous data in the database. But, the user is informed that the Article has been created successfully, even if it has not.

We need to tune our controller handling code so that it takes into account the failure of the statement `article.save`.

Here is an improved version of the `create` method that redirects user to the edit page of the article created, if `article.save` returns `true`. Otherwise, if `article.save` returns `false`, it renders the view template `app/views/articles/new.html.erb`, i.e. it asks the user to fill in the details again.

```
def create
  article = Article.new(article_params)
  if article.save
    redirect_to edit_article_url(article)
  else
    render :new
  end
end
```

(the above code snippet online)

Here is a short video that demonstrates the functionality of this implementation.

Blog Demo Articles Management First Part

The statement `redirect_to` is used to send back to the browser a `302` response with the URL the browser needs to take the user to. The URL that we are specifying here is the result of the call to `edit_article_url(article)` and it is more or less something like this: `http://localhost:3000/articles/<id of article created>/edit`.

You can also understand what is going on here, by reading, once more, the Rails logs:

```
Started POST "/articles" for 127.0.0.1 at 2017-05-29 12:49:19 +0300
Processing by ArticlesController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"kPuz61NgOf9Xjh+O8Vcv5jVz1GLmfV\
5i/gnfXZ//PocQf6Fj6ykr6agHxy7Sx6LgWgHlY0VKRW6CpWAiLHuaVg==", "article"=>{"title"=\
>"Ruby Hashes", "text"=>"The Ruby Hash API ..."}, "commit"=>"Save Article"}
   (0.1ms)  BEGIN
  SQL (0.3ms)  INSERT INTO "articles" ("title", "text", "created_at", "updated_at\
") VALUES ($1, $2, $3, $4) RETURNING "id"  [["title", "Ruby Hashes"], ["text", "T\
he Ruby Hash API ..."], ["created_at", "2017-05-29 09:49:19.122674"], ["updated_a\
```

```
t", "2017-05-29 09:49:19.122674"]]
   (0.2ms)  COMMIT
Redirected to http://localhost:3000/articles/10/edit
Completed 302 Found in 3ms (ActiveRecord: 0.6ms)


Started GET "/articles/10/edit" for 127.0.0.1 at 2017-05-29 12:49:19 +0300
Processing by ArticlesController#edit as HTML
  Parameters: {"id"=>"10"}
  Article Load (0.3ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id\
" = $1 LIMIT $2  [["id", 10], ["LIMIT", 1]]
  Rendering articles/edit.html.erb within layouts/application
  Rendered articles/edit.html.erb within layouts/application (0.5ms)
Completed 200 OK in 51ms (Views: 47.9ms | ActiveRecord: 0.3ms)
```

(the above code snippet online)

The first batch is the `POST` to create the article. Do you see how this HTTP Request terminates? It terminates with a `Redirected 302 to http://locahost:3000/articles/10/edit`. And you can see the `Completed 302 Found` HTTP response returned.

Then, the second batch of log lines, is about the request that the browser did to the URL the redirection specified, i.e. a `GET` to the `/articles/10/edit`.

But, what happens if the article cannot be saved? The `render :new` statement takes place. And the following video demonstrates that:

Blog Demo Articles Management First Part

Improving Error Handling

As you can see from the video, there are two problems:

1. The user is not actually informed about the error.
2. The user has lost any information that they might have entered. For example, if they have forgotten the `text` part which is mandatory, but they have only given the `title` part, the form is rerendered and the `title` value has been lost. So, user is required to enter again all the information from the beginning.

Can we fix these two issues?

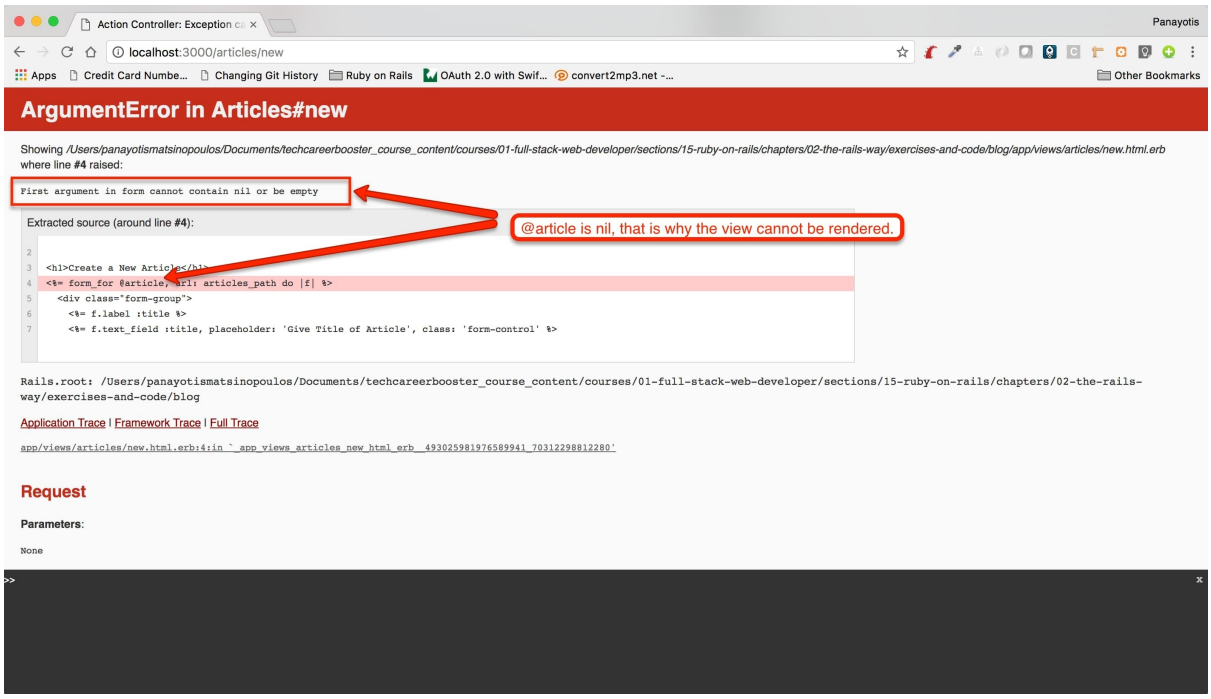
Let's start from the second problem. In order for the form to preserve the values that the user has submitted, we need to make the form rely on an instance of an Article. This instance will have to have the values that the user has submitted. There is a small change in the form code:

```
1  <% # File: app/views/articles/new.html.erb %>
2
3  <h1>Create a New Article</h1>
4  <%= form_for @article, url: articles_path do |f| %>
5    <div class="form-group">
6      <%= f.label :title %>
7      <%= f.text_field :title, placeholder: 'Give Title of Article', class: 'form-c\
8  ontrol' %>
9    </div>
10
11   <div class="form-group">
12     <%= f.label :text %>
13     <%= f.text_area :text, placeholder: 'Type in the text of the Article', class:\
14  'form-control' %>
15   </div>
16
17   <div>
18     <%= f.submit class: 'btn btn-success' %>
19   </div>
20 <% end %>
21
22 <div class="text-right">
23   <a href="<%= articles_path %>">List of Articles</a>
24 </div>
```

(the above code snippet online)

This is the same as before, but the only difference is on line 4. Instead of `form_for :article` we use `form_for @article` and we now have the form using an instance variable `@article` to build the input controls and its values from.

If you save the above and reload the http://localhost:3000/articles/new page on your browser, you will see this:



Error Because @article Is Nil

We cannot give `nil` to the `form_for` call. And `@article` is `nil` because we have not instantiated it.

Where do we instantiate the variables that the views need? In the corresponding controller actions. Which is the action that we deal with when we fetch the form to create a new Article? It is the `new` action. Having said that, we need to amend the `new` action to instantiate an Article and store it in the `@article` instance variable:

This is the version of the `new` action in our `ArticlesController`:

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

(the above code snippet online)

Now, reload the http://localhost:3000/articles/new page. You will see that the page is being loaded without any problem.

Now, if you try to save a new Article, it will succeed. But if you try to save an incomplete Article things will not go as expected. You will get the same error about providing `form_for` with `nil`. This is because the line 13, in Articles controller (`render :new`) asks Rails to render the `app/views/articles/new.html.erb` which requires the `@article` instance. And `create` action does not instantiate that.

This is the new version of `create` that will do that:

```
def create
  article = Article.new(article_params)
  if article.save
    redirect_to edit_article_url(article)
  else
    @article = article
    render :new
  end
end
```

(the above code snippet online)

The error branch (`else ... end`) will instantiate the `@article` to be equal to the `article` variable that has been created at the beginning of the method. Hence, the view `app/views/articles/new.html.erb` will, first, have the `@article` instantiated, and, second, will use the values of this instance to prepopulate the values of the input controls.

Now, when you submit and invalid Article, the values of the fields that you have completed are filled in with the values that you have given.

Here is how it works:

Values Are Retained When We Submit Invalid Article

Informing User

The other issue that we have to fix is about telling user that there was something wrong with the data that they have submitted. In order to address this one, we are going to enhance our `app/views/articles/new.html.erb` content to include markup that would display any errors if errors were present:

```
<% # File: app/views/articles/new.html.erb %>

<h1>Create a New Article</h1>
<%= 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>
```

```
18    <% end %>
19
20    <div class="form-group">
21      <%= f.label :title %>
22      <%= f.text_field :title, placeholder: 'Give Title of Article', class: 'form-c\
23 ontrol' %>
24    </div>
25
26    <div class="form-group">
27      <%= f.label :text %>
28      <%= f.text_area :text, placeholder: 'Type in the text of the Article', class:\
29  'form-control' %>
30    </div>
31
32    <div>
33      <%= f.submit class: 'btn btn-success' %>
34    </div>
35  <% end %>
36
37  <div class="text-right">
38    <a href="<%= articles_path %>">List of Articles</a>
39  </div>
```

(the above code snippet online)

The amendment is the block of code between lines 6 and 18. It is a `div` with id `error-explanation`, which is only printed if the instance `@article` has any errors attached. If it has...

1. We print the number of errors. Line 9. The `pluralize` method displays either `error` or `errors` depending on the number of errors the instance has. The number of errors is `@article.errors.count`.
2. For each one of the error messages (`@article.errors.full_messages.each`) we display a list item inside a `ul` tag. The list item contains the actual error message.

Let's try this. Try to create an Article with missing information, either with missing title or with missing text, or with both. See how the error message is displayed.

Watch the following video to see this in action:

Values Are Retained When We Submit Invalid Article

Style Div With Error Messages

Before we close this paragraph on error handling, you may want to introduce some kind of styling, in order to make the errors being displayed nicer.

One way you can do that is by creating the file `app/assets/stylesheets/errors.scss` with the following content:

```
/* File: app/assets/stylesheets/errors.scss */

#error-explanation {
  h2 { font-size: 1.4rem; }
}
```

(the above code snippet online)

Then, make sure that this file is being imported by your `app/assets/stylesheets/application.scss` file:

```
// File: app/assets/stylesheets/application.scss
//
@import "bootstrap-sprockets";
@import "bootstrap";
@import "errors";
```

(the above code snippet online)

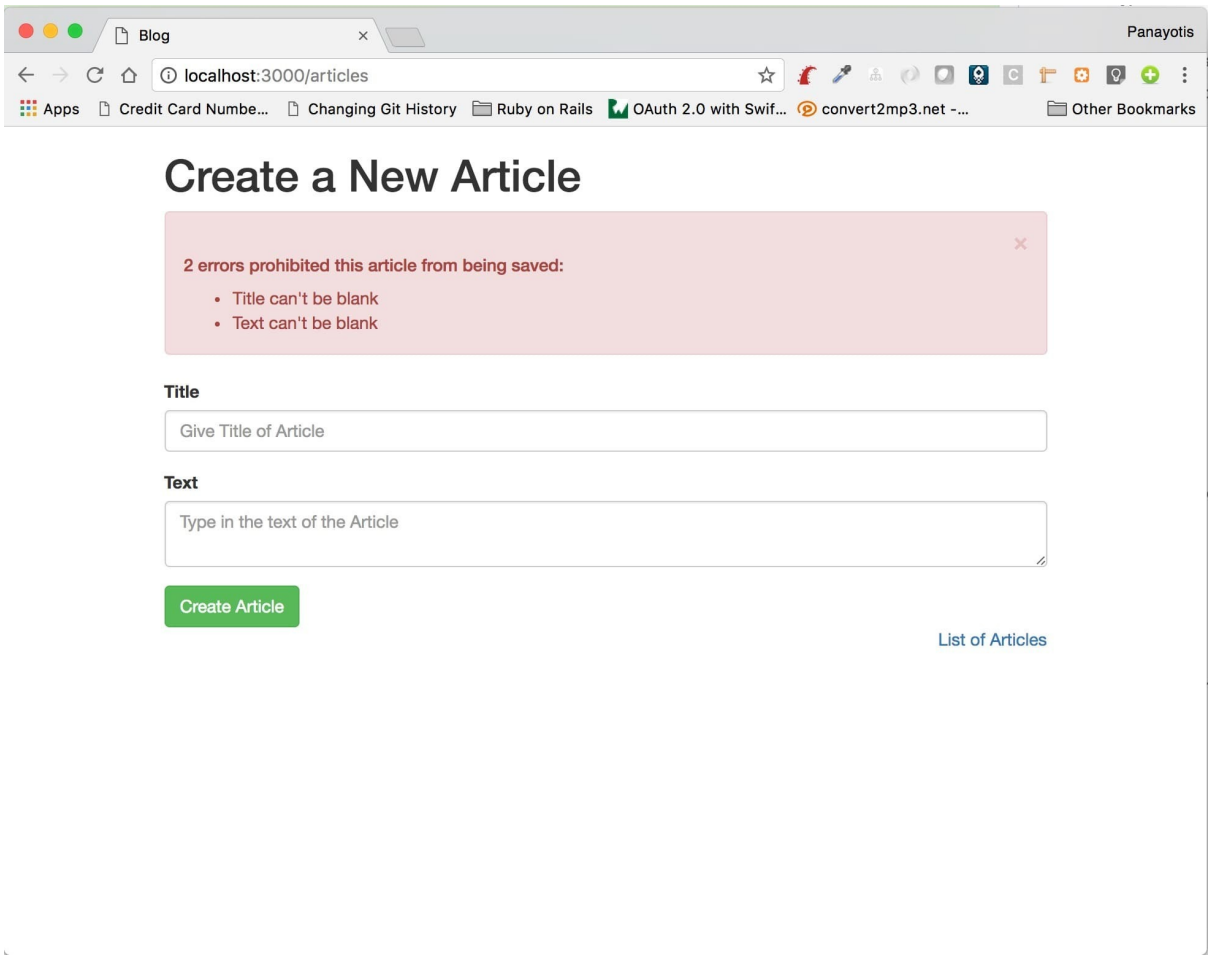
Also, enhance the markup that has to do with the `error-explanation` to have classes that are coming from Twitter Bootstrap, as follows:

```
...
<div id="error-explanation" class="alert alert-danger alert-dismissible" role="al\
ert">
  <button type="button" class="close" data-dismiss="alert" aria-label="Close"><sp\
an aria-hidden="true">&times;</span></button>
...
```

(the above code snippet online)

As you can see, we have added the classes `alert alert-danger alert-dismissible`. We have also added a `button` that will be used by the user to dismiss the alert.

Now, when you try to create an invalid Article, you will see something like this:



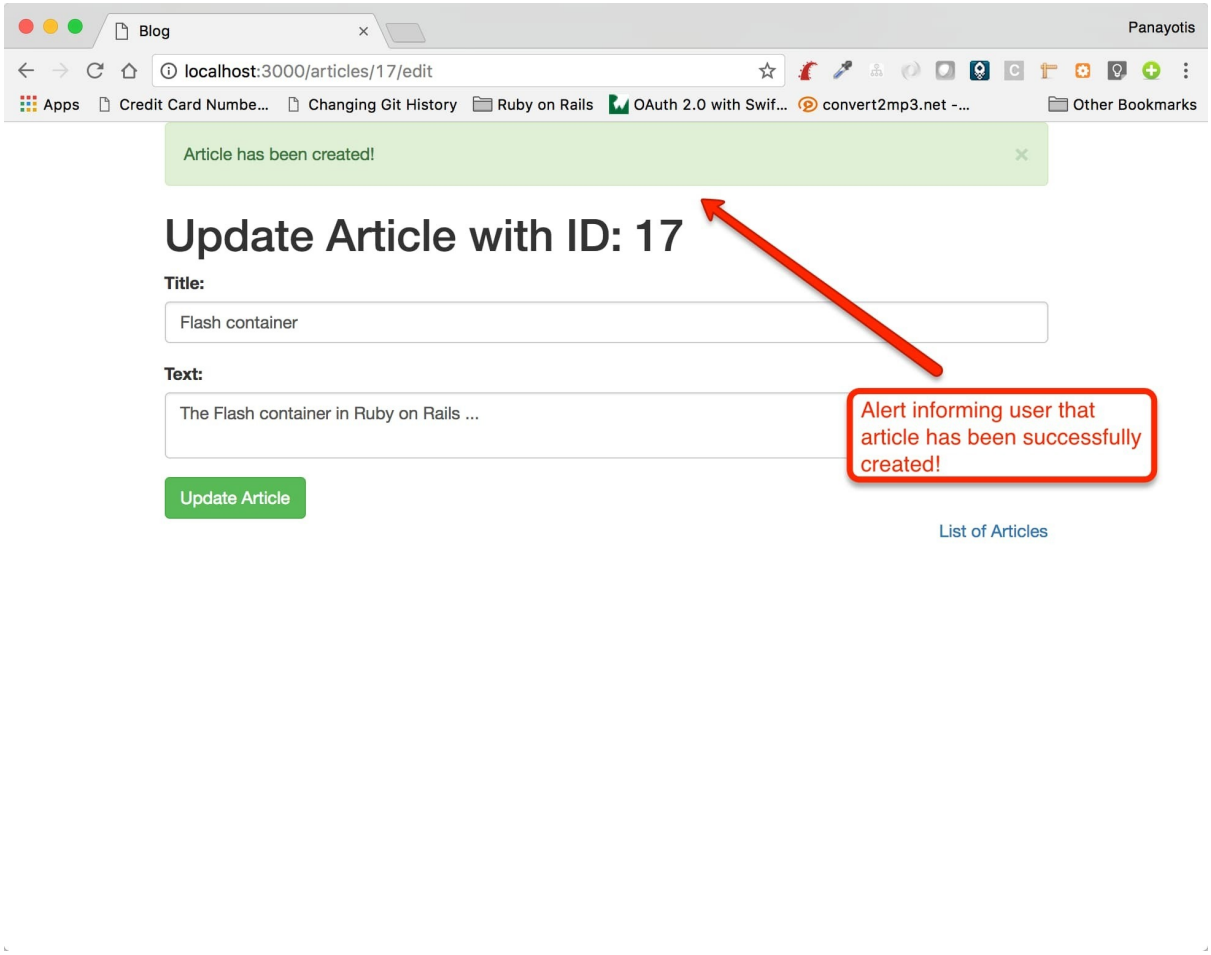
Style Container With Error Messages

Removing `create.html.erb` Confirmation Page

Since we don't display the confirmation page for the creation of an Article, we can now safely remove the `app/views/articles/create.html.erb` file.

Display Success Message

We have implemented the `create` method in such a way so that it redirects to the edit page of the article that has just been created. However, sometimes, it is also user-friendly, to be able to display an alert message that the creation was successful. Like this:



Informing User With an Alert That Article Has Been Created

Let's do that.

First, you need to prepare the message to display. This is done inside the `create` action of our Articles controller. Here is the implementation:

```
def create
  article = Article.new(article_params)
  if article.save
    flash[:success] = 'Article has been created!'
    redirect_to edit_article_url(article)
  else
    @article = article
    render :new
  end
end
```

(the above code snippet online)

The line that does the trick to prepare the message is line `flash[:success] = 'Article has been created!'`. The `flash` bucket is a Hash-like bucket that keeps its content **only for the next request**. Hence, It will be available when our server will process the `GET /articles/:id/edit`

request which will come next (thanks to the `redirect_to` statement). We could have used the `session` bucket instead (like we have done this when we worked with Sinatra), but for this case here, we prefer to use `flash` because `flash` lives only for the next request and we don't have to clear the bucket, like we would had to do if we were to use `session` instead.

Now that `flash` contains the message to display when processing the `GET /articles/:id/edit` request, we can read its content and display the message inside the `app/views/articles/edit.html.erb` which is the view that displays the edit form.

```
<% # File: app/views/articles/edit.html.erb %>

<% flash.each do |name, message| %>
  <div class="alert alert-<%= name %> alert-dismissible">
    <button type="button" class="close" data-dismiss="alert" aria-label="Close">&lt;\
span aria-hidden="true">&times;</span></button>
    <%= message %>
  </div>
<% end %>


<h1>Update Article with ID: <%= @article.id %></h1>

<form method="post" action="/articles/<%= @article.id %>">
...
```

(the above code snippet online)

Above, you can see the `<% flash ... end %>` block that we have added at the beginning of the file `app/views/articles/edit.html.erb`. The `flash`, is a `Hash` like bucket and can contain many keys. The above block creates one Twitter Bootstrap alert for each one of the keys. Note that the key is used to actually set the `alert-xxxxx` class that we attach to the `alert` used. This is a common practice with Twitter Bootstrap. Hence, when the key is `success`, like it does in our case, the class attached to the alert will be `alert-success`. The other values that can be used to create Twitter Bootstrap alerts with different semantics can be found here.

Save the changes in your `app/views/articles/edit.html.erb` and try to create a new valid Article. You will see that the success alert is being displayed.

Using Layout For `flash`

Usually, you will see the `flash` being rendered from the layout and not from a particular view. This is useful, because it allows all the views to use the `flash` bucket. Let's move the `flash` from the `app/views/articles/edit.html.erb` to the `app/views/layouts/application.html.erb`:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Blog</title>
5      <%= csrf_meta_tags %>
6
7      <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-trac\
8  k': 'reload' %>
9      <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
10   </head>
11
12   <body>
13     <div class="container">
14
15       <% flash.each do |name, message| %>
16           <div class="alert alert-<%= name %> alert-dismissible">
17             <button type="button" class="close" data-dismiss="alert" aria-label="\
18 Close"><span aria-hidden="true">&times;</span></button>
19             <%= message %>
20           </div>
21       <% end %>
22
23       <%= yield %>
24
25     </div>
26   </body>
27 </html>
```

(the above code snippet online)

You can see the lines 14 to 19 that display the `flash` like we did in the `app/views/articles/edit.html.erb`. If you try to create the Article again, you will see the `flash` being displayed as before.

Anchors With `link_to`

There is another helper method that Rails provides us with, and it has to do with creating anchor HTML tags. This is the `link_to` method.

Let's take for example the `app/views/articles/index.html.erb` view template. We are going to replace the anchor tags with the use of this method.

```
4
5   <table class="table table-striped">
6     <thead>
7       <tr><th>ID</th><th>Title</th><th>Text</th><th></th></tr>
8     </thead>
9     <tbody>
10      <% @articles.each do |article| %>
11        <tr>
12          <td><%= article.id %></td>
13          <td><%= article.title %></td>
14          <td><%= article.text %></td>
15          <td>
16            <a href="/articles/<%= article.id%>">Show</a>  
17            <a href="/articles/<%= article.id%>/edit">Edit</a>   
18            <form method="post" action="/articles/<%= article.id %>" style="...">
19              <input type="hidden" name="_method" value="delete"/>
20              <input type="hidden" name="authenticity_token" value="<%= form_authenticity_token %>"/>
21              <button type="submit" class="btn btn-danger">Delete</button>
22            </form>
23          </td>
24        </tr>
25      <% end %>
26    </tbody>
27  </table>
28  <div>
29    <a href="/articles/new">Create Another Article</a>
30  </div>
31
```

anchors to change to link_to

Articles Index - Anchors to change to link_to

The `link_to` usually works in conjunction with the path and url helpers that we saw earlier. Here are the changes that we are going to do:

1. `<a href="/articles/<%= article.id%>">Show</a>` is going to be replaced with `<%= link_to 'Show', article_path(article) %>`.
2. `<a href="/articles/<%= article.id%>/edit">Edit</a>` is going to be replaced with `<%= link_to 'Edit', edit_article_path(article) %>`.
3. `<a href="/articles/new">Create Another Article</a>` is going to be replaced with `<%= link_to 'Create Another Article', new_article_path %>`.

Do the above changes and visit the page http://localhost:3000/articles. If you view the source of the page, you will see that the anchors have been generated by the `link_to` statements.

Generating the anchor tags with the help of the `link_to` command, in conjunction with the path helpers is much more safe and powerful. Also, the `html.erb` file is much easier to read.

Delete Links

In file `app/views/articles/index.html.erb` we have used a `form` to build the markup for the `Delete` action on each article. An alternative would have been to have used an anchor instead.

Replace the code:

```
<form method="post" action="/articles/<%= article.id %>" style="display: inline;">
  <input type="hidden" name="_method" value="delete"/>
  <input type="hidden" name="authenticity_token" value="<%= form_authenticity_tok\
en %>"/>
  <button type="submit" class="btn btn-danger">Delete</button>
</form>
```

(the above code snippet online)

with the following:

```
<%= link_to 'Delete', article_path(article), method: :delete, class: 'btn btn-dan\
ger' %>
```

(the above code snippet online)

Now, save the file and visit the http://localhost:3000/articles. You will still see the `Delete` button as one of the available actions on each Article entry in the table. However, now, there is no `form` in the markup, but there is a anchor instead. If you inspect the source on the `Delete` element, you will see that the following markup is generated:

```
<a class="btn btn-danger" rel="nofollow" data-method="delete" href="/articles/17"\
>Delete</a>
```

(the above code snippet online)

1. The anchor has the `rel` attribute set to `nofollow`. This is very important, otherwise, the crawling robots would *click* on this link.
2. There is a `data` attribute with name `data-method` and value `delete`. This is to tell browser that we want to send over a `DELETE` HTTP request.
3. The `href` value of the anchor points to the article that we want to delete.

Note that there is some JavaScript involved here for the actual submission of the HTTP request. The JavaScript provided by the Rails framework and it is automatically invoked when we click on the particular link.

Delete With Confirmation

Let's now add a feature on our `Delete` buttons. We will have the browser ask the user whether they really want to delete the Article or not.

Go to line that has the link to delete, `<%= link_to 'Delete', article_path(article), method: :delete, class: 'btn btn-danger' %>` and add the data attribute `confirm` with the question that you want the user to be asked. In other words, your `link_to` needs to be like this:

```
<%= link_to 'Delete', article_path(article), method: :delete, class: 'btn btn-dan\
ger', data: {confirm: 'Are you sure that you want to delete this Article?'} %>
```

(the above code snippet online)

Now, save your view template code and reload the http://localhost:3000/articles page. Then try to delete an article. You will see that the browser is asking for a confirmation, with the question that you have just added as value to the data attribute `confirm`.

Also, inspect the code that is generated for this link:

```
<a class="btn btn-danger" data-confirm="Are you sure that you want to delete this\
 Article?" rel="nofollow" data-method="delete" href="/articles/13">Delete</a>
```

(the above code snippet online)

Do you see the `data-confirm` HTML attribute? This is the result of the `data: {confirm: 'Are you sure that you want to delete this Article?'}` that we have added to the `link_to` call.

> *Information*: This is a standard way to add `data` HTML attributes in Rails views. `data: {x: y}` is adding the data attribute with name `data-x` and value `y`.

Using `before` Hooks In Controller

Sometimes, we want to reduce code duplication in our controllers' code. One way you can do that is by factoring out into a method, the code that is commonly used in more than one action.

For example, we can see that the actions `show`, `edit`, `update` and `destroy`, the first thing they do, is to locate the Article that the `params[:id]` specifies. We can tell controller that these actions need to execute the same common piece of code *before* the actual action code.

This is the new version of the Articles controller that does that:

```ruby
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  before_action :find_article, only: [:show, :edit, :update, :destroy]

  def new
    @article = Article.new
  end

  def create
    article = Article.new(article_params)
    if article.save
      flash[:success] = 'Article has been created!'
      redirect_to edit_article_url(article)
    else
      @article = article
      render :new
    end
  end

  def show
  end

  def index
    @articles = Article.all
```

```
26    end
27
28    def edit
29    end
30
31    def update
32      @article.title = params[:title]
33      @article.text = params[:text]
34
35      @article.save
36    end
37
38    def destroy
39      @article.destroy
40    end
41
42    private
43
44    def article_params
45      params.require(:article).permit(:title, :text)
46    end
47
48    def find_article
49      @article = Article.find(params[:id])
50    end
51  end
```

(the above code snippet online)

You can see that in lines 48 to 50 we have defined the private method `find_article`. This locates the Article using the `params[:id]` and stores the retrieved Article inside the instance variable `@article`.

Since this functionality needs to be executed *before* the actions `show`, `edit`, `update` and `destroy`, we have added the code on line 4:

```
1  before_action :find_article, only: [:show, :edit, :update, :destroy]
```

(the above code snippet online)

And we have removed the `@article = Article.find(params[:id])` piece of code, from the actions `show`, `edit`, `update` and `destroy` implementation.

Now, Rails knows that it will execute the `find_article` method before the actual execution of the action for each one of the actions `show`, `edit`, `update` and `destroy`.

Save the above code and experiment with the actions `show`, `edit`, `update` and `destroy`. You will see that they will be functioning correctly, as before.

Also, now that `show` and `edit` actions are empty, we can completely remove them from the `ArticlesController` code.

```ruby
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  before_action :find_article, only: [:show, :edit, :update, :destroy]

  def new
    @article = Article.new
  end

  def create
    article = Article.new(article_params)
    if article.save
      flash[:success] = 'Article has been created!'
      redirect_to edit_article_url(article)
    else
      @article = article
      render :new
    end
  end

  def index
    @articles = Article.all
  end

  def update
    @article.title = params[:title]
    @article.text = params[:text]

    @article.save
  end

  def destroy
    @article.destroy
  end

  private

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

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

(the above code snippet online)

Using Partials

View templates can become quite long. Also, parts of your view templates code might be the same from one view template to the next. Rails provides you with a mechanism to share code between view templates and layouts. It is called `partials`.

We are going to apply this technique in the `app/views/layouts/application.html.erb`. We will factor out in its own partial the code that has to do with the `flash`.

Let's create the file `app/views/layouts/_flash.html.erb`. Note that the filename is `_flash.html.erb` and not `flash.html.erb`. It has a prefix _ and this is how we should name the `partials` in Rails.

Then, remove the `flash` part from the `app/views/layouts/application.html.erb` and put it inside this new file:

```
<% # File: app/views/layouts/_flash.html.erb %>

<% flash.each do |name, message| %>
  <div class="alert alert-<%= name %> alert-dismissible">
    <button type="button" class="close" data-dismiss="alert" aria-label="Close">&lt;\
span aria-hidden="true">&times;</span></button>
    <%= message %>
  </div>
<% end %>
```

(the above code snippet online)

Now, add the line `<%= render partial: 'layouts/flash' %>` in your `app/views/layouts/application.html.erb` file, at the point you previously had the flash code. Hence, your `app/views/layouts/application.html.erb` needs to be this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Blog</title>
    <%= csrf_meta_tags %>

    <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-trac\
k': 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <div class="container">

      <%= render partial: 'layouts/flash' %>
```

```
16
17        <%= yield %>
18
19      </div>
20    </body>
21  </html>
```

(the above code snippet online)

Do you see line 14? This is the replacement for the flash code which has now been moved inside the `app/views/layouts/_flash.html.erb` file. The call to `render` takes as argument the partial to render. The argument is given as a Hash with the key `:partial`. Note that we specify the partial using a relative path to the `app/views` folder. Also, we do not specify the _ that is the actual character the partial filename starts with.

Rails Console

Now, I will introduce you to a very useful tool that is provided by Rails. The Rails Console. Similar to the Ruby console, the Rails Console gives you a non-UI interface to your Web application. It can be started with `bin/rails console`. Let's do it:

```
1  blog $ bin/rails console
2  Loading development environment (Rails 5.1.1)
3  2.3.3 :001 >
```

(the above code snippet online)

You can see the > prompt. It allows you to issue any Rails command that you like. For example, you can count the number of Articles with the following command:

```
1  2.3.3 :001 > Article.count
2     (0.6ms)  SELECT COUNT(*) FROM "articles"
3   => 4
4  2.3.3 :002 >
```

(the above code snippet online)

You can even do updates in your database.

```
2.3.3 :002 > a = Article.first
  Article Load (0.3ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
"id" ASC LIMIT $1  [["LIMIT", 1]]
 => #<Article id: 1, title: "Of Mice And Men", text: "Of Mice and Men text", crea\
ted_at: "2017-05-28 09:18:29", updated_at: "2017-05-28 09:18:29">
2.3.3 :003 > a.text = "Of Mice and Men is a book written by John Steinbeck"
 => "Of Mice and Men is a book written by John Steinbeck"
2.3.3 :004 > a.save
   (0.2ms)  BEGIN
  SQL (0.6ms)  UPDATE "articles" SET "text" = $1, "updated_at" = $2 WHERE "articl\
es"."id" = $3  [["text", "Of Mice and Men is a book written by John Steinbeck"], \
["updated_at", "2017-05-30 08:11:30.249219"], ["id", 1]]
   (1.0ms)  COMMIT
 => true
2.3.3 :005 >
```

(the above code snippet online)

In the above, we locate the first article (`a = Article.first`) and then we update its `text` attribute and we save the update in the database.

Rails console is very useful when doing development and sometimes it can be useful when working on a production instance, although in this latter case you need to be very careful because you might change data in your production database.

Exit from the Rails console by giving `exit`.

HAML - HTML Abstraction Markup Language

Before we close this chapter with refactorings that made our code being more Railsish, I would like to introduce you to HAML. HAML stands for *HTML Abstraction Markup Language* and its intention is to help us write less code within our view templates.

Look for example how this piece of code here, that is using ERB:

```
<div class="container">
  <h1><%= post.title %></h1>
  <h2><%= post.subtitle %></h2>
  <div class="content">
    <%= post.content %>
  </div>
</div>
```

(the above code snippet online)

can now be written with HAML:

```
.container
  %h1= post.title
  %h2= post.subtitle
  .content
    = post.content
```

(the above code snippet online)

At the beginning, it might look a little bit strange, but, when you get used to it, you will see how powerful it is and how cleaner your view templates code becomes.

Let's Introduce HAML Gem

In order to use HAML in your Rails project, we are recommending using the `haml-rails` gem. Hence, add this gem to your Gemfile:

```
gem 'haml-rails'
```

(the above code snippet online)

and run `bundle` to install HAML gems.

Convert the ERB files to HAML

Now that you have installed the HAML related gem, you have at your disposal a `rake` task which can convert all your `.erb` files to `.haml` files. Execute the following command:

```
blog $ bin/rake haml:erb2haml
--------------------------------------------------------------------------------
Generating HAML for app/views/articles/destroy.html.erb...
Generating HAML for app/views/articles/edit.html.erb...
Generating HAML for app/views/articles/index.html.erb...
Generating HAML for app/views/articles/new.html.erb...
Generating HAML for app/views/articles/show.html.erb...
Generating HAML for app/views/articles/update.html.erb...
Generating HAML for app/views/layouts/_flash.html.erb...
Generating HAML for app/views/layouts/application.html.erb...
Generating HAML for app/views/layouts/mailer.html.erb...
Generating HAML for app/views/layouts/mailer.text.erb...
Generating HAML for app/views/welcome/index.html.erb...
--------------------------------------------------------------------------------
HAML generated for the following files:
        app/views/articles/destroy.html.erb
        app/views/articles/edit.html.erb
        app/views/articles/index.html.erb
        app/views/articles/new.html.erb
        app/views/articles/show.html.erb
```

```
        app/views/articles/update.html.erb
        app/views/layouts/_flash.html.erb
        app/views/layouts/application.html.erb
        app/views/layouts/mailer.html.erb
        app/views/layouts/mailer.text.erb
        app/views/welcome/index.html.erb
-------------------------------------------------------------------------------
Would you like to delete the original .erb files? (This is not recommended unless\
 you are under version control.) (y/n)
y
Deleting original .erb files.
-------------------------------------------------------------------------------
Task complete!
No .erb files found. Task will now exit.
blog $
```

(the above code snippet online)

> As you can see, I have answered `y` to the question `Would you like to delete the original .erb files?`.

Now, all your files have been converted to HAML files. Let's take for example, the file `app/views/articles/new.html.haml`, which is the HAML version of the original `app/views/articles/new.html` file:

```
- # File: app/views/articles/new.html.erb

%h1 Create a New Article

= form_for @article, url: articles_path do |f|
  - if @article.errors.any?
    #error-explanation.alert.alert-danger.alert-dismissible{:role => "alert"}
      %button.close{"aria-label" => "Close", "data-dismiss" => "alert", :type => \
"button"}
        %span{"aria-hidden" => "true"} ×
      %h2
        = pluralize(@article.errors.count, "error")
        prohibited this article from being saved:
      %ul
        - @article.errors.full_messages.each do |msg|
          %li= msg
  .form-group
    = f.label :title
    = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-con\
trol'
  .form-group
```

```
22      = f.label :text
23      = f.text_area :text, placeholder: 'Type in the text of the Article', class: '\
24  form-control'
25    %div
26      = f.submit class: 'btn btn-success'
27
28  .text-right
29    %a{:href => articles_path} List of Articles
```

(the above code snippet online)

Here are the basic HAML rules that can help you understand what is going on in the above content:

Indentation

HAML organizes its code in blocks, each one possibly nested into another. The indentation used to nest a block inside another is two spaces.

For example, line 5 starts a block that ends on line 24. Actually, its last line is line 23. The block starts on line 5 at column 1 and all its content is indented two spaces to the right, column 3.

Nested Blocks

A nested block can contain other nested blocks. For example, the block at line 5 till 24 contains three other blocks.

1. First block are lines 6 to 15
2. Second block are lines 16 to 18
3. Third block are lines 19 to 21
4. Fourth block are lines 22 to 23

Every time we nest a new block inside another, we have to indent by two spaces.

Ruby Code

You can write Ruby code inside the view templates like you did with ERB when you enclosed the Ruby code in blocks of `<% %>`

For example, line 1 has one line of Ruby code:

```
1  - # File: app/views/articles/new.html.erb
```

(the above code snippet online)

which is equivalent to

```
<% # File: app/views/articles/new.html.erb %>
```

(the above code snippet online)

Output to HTML Buffer

When you want your Ruby code to send its output to the HTML buffer, you have to use the = and not the - symbol. In other words, the `<%=` has the equivalent of = in HAML.

This is how the line 5 generates the `form` HTML tag. The `<%= form_for ...` has been converted to `= form_for ....`

No Need To End Blocks

With HAML you don't have to *end* the blocks, like you did with ERB.

So, this:

```
<% if @article.errors.any? %>
...
<% end %>
```

(the above code snippet online)

is simply written:

```
- if @article.errors.any?
```

(the above code snippet online)

This is because of the nesting and indentation rules that help HAML parser find the end of the block without the need for the `end` markers.

HTML Tags

All the HTML tags in HAML need to start with the `%` symbol and then the tag name. For example, on line 22, we create a `div` tag:

```
%div
  = f.submit class: 'btn btn-success'
```

(the above code snippet online)

The HTML content that goes inside the HTML tag needs to be nested in the following line(s) using two spaces indentation.

`div` Tags with Ids

`div` tags that they have an id, they can be written with the id following a `#` symbol, without the `%div` to be used. In other words the following are equivalent:

1. `%div{'id' => 'error-explanation'}`
2. `%div#error-explanation`
3. `#error-explanation`

The last option is the one that we are using on line 7.

`div` Tags with Classes

Also, if a `div` has at least one class, then we can start with the class without specifying explicitly that it is about a `div`. In other words, the following are equivalent:

1. `%div{class: 'form-group'}`
2. `%div.form-group`
3. `.form-group`

And we can have multiple classes on the same `div`, one listed after the other.

We have used this technique multiple times in the `app/views/articles/new.html.erb` file. See lines 7, 16, 19 and others.

HTML Attributes

You can attach HTML attributes to your HTML tags by just listing them into a `Hash` next to your HTML tag definition. This is how we have done it on line 7, for example, for the `role` HTML attribute. The good think with this approach is that your `Hash` is Ruby code and you can use any Ruby code to build its keys and/or its values.

Content On Same Line

Sometimes, we give the HTML content of a tag on the same line with the line that we open the tag with. Look at line 15 for example. The

```
1 %li= msg
```

(the above code snippet online)

is equivalent to

```
%li
  = msg
```

(the above code snippet online)

We prefer the one line approach when the block content is actually only one line. Hence, here, we prefer the `%li= msg` version.

HAML Reference

HAML offers a lot more features than the ones we have seen above. After you get a little bit familiar with HAML, we suggest that you read the HAML Reference. At least once. So that you know what this tool can offer to you.

Online HTML To HAML Conversion

I sometimes find handy this HTML 2 HAML Conversion tool. You may find it handy too.

Closing Note

That was a second round on Rails features that make our life much easier when developing a Web Application. Definitely, this is not an exhaustive list of the techniques and tools provided by Rails. We will learn more in the following chapters.

Important Don't forget to commit your code and push to your Github remote repository. Also, deploy your changes to your production Heroku application.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

Task A

You need to finish the `blog` polishing. You need to apply all the things you have learned so far at all parts of the code.

You will have to enhance the following points, at least:

1. The `update` action of Articles controller. Use the `Article#update` ActiveRecord method and the `article_params` method that you have already created. The `Article#update` takes as input a `Hash` with the attributes/values to update.

2. Make sure that the `update` action handles the errors. `Article#update` will return `false` if something goes wrong.
3. `update` needs to redirect back to `edit_article_path`, in case of success. It needs to display an alert that update has been carried out successfully.
4. Change the `edit` view template to use Rails form helper methods. Note that the `form_for` needs to define the correct `method` value for the submission of an update.
5. When `update` fails it needs to display an alert with the errors.
6. Change the `destroy` action to redirect to the list of articles, if destroy succeeds. Also, it needs to display an alert that the destroy succeeded.
7. Make sure that all the anchors are built using the `link_to` method.

Make any other improvement you see fit.

Important Push your work to your Github account and to your production Heroku server.

Task B

In the previous chapter you have started implementing an e-shop. Apply all the things that you have learned in this chapter to this project.

Important Push your work to your Github account and to your production Heroku server.

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

3 - Authentication

Summary

Rails Authentication

Authentication is very popular feature in many Web applications. It is useful to protect functions of the Web application that should only be used by authenticated users. With authentication we know who is the user that is trying to carry out an action and we either allow or prevent these actions from taking place. In this chapter we work the Basic Authentication and the Session-Based Authentication.

Learning Goals

1. Learn what Authentication is.
2. Learn about the Basic Authentication.
3. Learn about Session Based Authentication.
4. Learn how you can encrypt Users' password.
5. Learn how to define validations rules that make values unique.
6. Learn how to model a User with email and password.
7. Learn how you can use a password confirmation to double check password given to create a User.
8. Learn how you can create a Sign Up page.
9. Learn how you can create a Sign In page.
10. Learn how to create non-persisting Models.
11. Learn how to write custom validations.
12. Learn how to keep track of the currently signed in user.

13. Learn how to define helper methods inside the controllers.
14. Learn how to implement the Sign Out functionality.
15. Learn how to make the actions in a controller consistent to the application logic.
16. Learn how to reduce controller code duplication using filters.
17. Learn how to protect some of your pages from being publicly accessed.
18. Learn how you can redirect user back to a originally requested private page after signing in.
19. Learn how to hide links that are not applicable.
20. Learn how to minimize code duplication in your views, using partials.
21. Learn how to pass real arguments to populate the locals of a partial.
22. Learn how form properties can be derived from the model state.

Introduction

Authentication is the feature used to identify the user that is trying to carry out an action. Our `blog` application allows anyone to access it, but, this wouldn't have been the case in a real blog production application. Can we protect the resources of our `blog` according to who is trying to access them.

Basic Authentication

Let's start with the easiest authentication mechanism that Rails offers. It is the Basic Authentication. If you want to read about Basic Authentication you can read this post here or google for it. In simple words, we guard our server in such a way so that when a browser tries to access our Web application resources, the browser is obliged to provide a *username* and a *password*, credentials that are usually provided by the real user behind the browser.

Our `blog` Basic Authentication

We want to protect our `blog` application with basic authentication. In fact:

1. We want the pages that have to do with creating, updating or deleting content to be authenticated.
2. The pages that list or show blog posts should not be authenticated, but they should be free to everyone.

![Basic Authentication Demo]images/sections/15-ruby-on-rails/chapters/03-authentication/blog-basic-authentication-demo-page.jpg)

Let's do that. Open the `app/controllers/articles_controller.rb` and amend the beginning of the class implementation with the following line:

```
http_basic_authenticate_with name: ENV['BASIC_AUTHENTICATION_USERNAME'], password\
: ENV['BASIC_AUTHENTICATION_PASSWORD'], except: [:index, :show]
```

(the above code snippet online)

In other words, the `ArticlesController` class starts with code like this:

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  http_basic_authenticate_with name: ENV['BASIC_AUTHENTICATION_USERNAME'], passwo\
rd: ENV['BASIC_AUTHENTICATION_PASSWORD']

  before_action :find_article, only: [:show, :edit, :update, :destroy]

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

(the above code snippet online)

The `http_basic_authentication_with` is the method call that does the trick to enable Basic Authentication. The arguments to this method are the username and password, as well as a specification about the actions the Basic Authentication will be applied to.

The username is specified with the key `:name` and the password is specified with the key `:password`. The actions specification is optional. If missing, that would mean all the actions. Otherwise it can be either `:except`, with value being an array with the actions to exclude from Basic Authentication, or `:only`, with value being an array with the actions to include in the Basic Authentication.

For our `ArticlesController`, we want Basic Authentication for all the actions except `:index` and `:show`. It's better to use a white list approach here, i.e. apply Basic Authentication to all actions by default and specify the list that Basic Authentication is not listed, with the `:except` key, rather than follow the black list approach to allow all actions being free except the ones that you would specify with `:only`. With `:except`, we make sure that we don't expose a new action to the public accidentally. Whereas with `:only` we need to make sure that whenever we add a new action, to consider whether we want it to be Basic Authenticated or not.

Watch out how we specify the credentials. We do not hard code the values inside the `ArticlesController`. We will use environment variables. And because of the fact that we are using `dotenv-rails` gem, we are going to specify these values inside the `.env` file:

> **Important**: the `.env` and `.env.test` files are being git ignored because they have a corresponding entry inside the `.gitignore` file. This setting has been carried out in the previous chapter. If you have not done it, please go ahead and do it. Add the `.env` and `.env.test` names inside the `.gitignore` file.

```
BASIC_AUTHENTICATION_USERNAME=admin
BASIC_AUTHENTICATION_PASSWORD=admin
```

(the above code snippet online)

Having done this work, now let's try to visit the http://localhost:3000/articles and then the page http://localhost:3000/articles/new. The first page will be visited without any problem and without the browser asking for username and password. However, the second page will require us to give the username and the password. Try to give the username and the password that you have specified in the `.env` file and you will get to the page to create the new article. Also, other pages that have to do with creation, update or delete of an Article are now accessed without username and password, since browser remembers to keep on sending the same credentials while in the same session.

Basic Authentication Demo

> *Hint:* After you have authenticated successfully, if you want to experience the modal dialog asking username and password again, you will have to restart your browser. Or you can always start a new incognito / private window. In fact, working with incognito windows will be better. If you close the incognito window and start a new one, it will ask for username and password when you visit the authenticated pages.

The Basic Authentication is useful and easy to deploy. However, it has some disadvantages. For example, there is no easy way to clear the credentials. Browsers, cache the username and password for a proprietary amount of time, that differs from browser to browser. Also, the modal asking for username and password does not allow the integration with other methods of authentication, for example using social networks.

Session Based Authentication

We will now proceed with the implementation of another technique for authentication.

1. We will introduce a User model into our application. This will be used to store the users that have signed up to use our application.
2. Sign Up Page: We will build a special page that will be asking user to sign up.
3. Sign In Page: We will build a special page that will be allowing user to sign in.

Create User Model

`users` table

Let's start with the `users` table. We will initially generate the migration:

```
blog $ bin/rails generate migration create_users
      invoke  active_record
      create    db/migrate/20170604052426_create_users.rb
```

(the above code snippet online)

Let's put the following content inside the migration file:

```ruby
# File: db/migrate/20170604052426_create_users.rb
#
class CreateUsers < ActiveRecord::Migration[5.1]
  def change
    create_table :users do |t|
      t.string :email, null: false
      t.string :password_digest, null: false

      t.timestamps
    end
  end
end
```

(the above code snippet online)

As you can read from the above, the table will contain the email and the encrypted password (column `password_digest`) of the User. Note that we will not be saving the password in clear form. We will be encrypting it. The algorithm will be a one-way encryption algorithm that will prevent anyone from being able to discover the actual value.

Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170604052426 CreateUsers: migrating ======================================
-- create_table(:users)
   -> 0.0103s
== 20170604052426 CreateUsers: migrated (0.0104s) =============================
```

(the above code snippet online)

Now the table exists in our database.

Add `bcrypt` Gem

Edit your `Gemfile` and uncomment the line `# gem 'bcrypt', '~> 3.1.7'`. In other words, bring the `bcrypt` gem in. This will be used for the encryption of the User password. When you bring the `bcrypt` in, run `bundle` to install the gem.

```
blog $ bundle
...
Installing bcrypt 3.1.11 with native extensions
...
blog $
```

(the above code snippet online)

The `User` Model

Let's now create the file `app/models/user.rb`. This is going to have the model of the `User` as an `ApplicationRecord` sub-class:

```ruby
# File app/models/user.rb
#
class User < ApplicationRecord
  validates :email, presence: true, uniqueness: { case_sensitive: false }
  has_secure_password
end
```

(the above code snippet online)

The above model does the following:

1. It tells that the `email` attribute of a User needs to be present. Otherwise, the model will not be saved in the database. Note that Rails ActiveRecord does not save invalid models in the database.
2. Also, the `email` needs to be unique. This means that when a new User tries to be saved into the database but there is already another one with the same email, then this new User will not be saved and will be considered invalid. Note also the `case_sensitive: false` property of the validation. We are asking ActiveRecord to consider two emails the same, even if they differ in their case. In other words, `Foo@gmail.com` will be considered equal to `foo@GMAIL.com`, for example.
3. We call the class method `has_secure_password`, that does a lot of things for us.
   1. It makes sure that we cannot create a User without a value for `password_digest`.
   2. It populates the `password_digest` attribute, given the `password` value.
   3. It may also validate that a `password confirmation`, given alongside a `password`, has the same value as the `password`.
   4. Also, it makes sure that the `password` does not have length greater than 72 characters long.
   5. Offers an `#authenticate` method that will return `false` if the authentication is carried out against a `password` that does not match the user `password`.

Use the `User` Model

With the above model created, let's open the Rails console and experiment with the `User` model. It will be instructive and let us know how the model works.

```
blog $ bin/rails console
2.3.3 :001 > User.count
   (0.5ms)  SELECT COUNT(*) FROM "users"
 => 0
2.3.3 :002 > User.create! email: 'foo@gmail.com', password: '12345678'
   (0.2ms)  BEGIN
  User Exists (0.5ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "foo@gmail.com"], ["LIMIT", 1]]
  SQL (0.4ms)  INSERT INTO "users" ("email", "password_digest", "created_at", "up\
dated_at") VALUES ($1, $2, $3, $4) RETURNING "id"  [["email", "foo@gmail.com"], [\
"password_digest", "$2a$10$znMwmqMs1pUgorvFon9LQOk0gP4fcWkISEpx5CcJ.4IixuSWH4toK"\
], ["created_at", "2017-06-04 05:52:46.214002"], ["updated_at", "2017-06-04 05:52\
:46.214002"]]
   (2.2ms)  COMMIT
 => #<User id: 6, email: "foo@gmail.com", password_digest: "$2a$10$znMwmqMs1pUgor\
vFon9LQOk0gP4fcWkISEpx5CcJ.4I...", created_at: "2017-06-04 05:52:46", updated_at:\
 "2017-06-04 05:52:46">
2.3.3 :003 > User.create! email: 'foo@Gmail.com', password: '12345678'
   (0.2ms)  BEGIN
  User Exists (0.4ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "foo@Gmail.com"], ["LIMIT", 1]]
   (0.2ms)  ROLLBACK
ActiveRecord::RecordInvalid: Validation failed: Email has already been taken
        from (irb):3
2.3.3 :004 > User.count
   (0.6ms)  SELECT COUNT(*) FROM "users"
 => 1
2.3.3 :005 >
```

(the above code snippet online)

The above interaction confirms the `email` uniqueness with case insensitive comparison. You can see that when we tried to create the second User with email `foo@Gmail.com` failed because of the already existing User with `email foo@gmail.com`.

Also, see how the `password_digest` is automatically populated with an encrypted version of the password.

With the `rails console` still running, let's see the authentication API that has been attached to the `User` model:

```
2.3.3 :006 > u = User.find_by_email('foo@gmail.com')
  User Load (0.3ms)  SELECT  "users".* FROM "users" WHERE "users"."email" = $1 LI\
MIT $2  [["email", "foo@gmail.com"], ["LIMIT", 1]]
 => #<User id: 6, email: "foo@gmail.com", password_digest: "$2a$10$znMwmqMs1pUgor\
vFon9LQOk0gP4fcWkISEpx5CcJ.4I...", created_at: "2017-06-04 05:52:46", updated_at:\
 "2017-06-04 05:52:46">
2.3.3 :007 > u.authenticate('1234567')
 => false
2.3.3 :008 > u.authenticate('12345678')
 => #<User id: 6, email: "foo@gmail.com", password_digest: "$2a$10$znMwmqMs1pUgor\
vFon9LQOk0gP4fcWkISEpx5CcJ.4I...", created_at: "2017-06-04 05:52:46", updated_at:\
 "2017-06-04 05:52:46">
2.3.3 :009 >
```

(the above code snippet online)

See how the authentication works by studying the above lines. The first command locates the user by email.

```
u = User.find_by_email('foo@gmail.com')
```

(the above code snippet online)

The second command calls the `#authenticate` method with `1234567`, which is not the password of the user loaded in `u`. Remember that the password that we used to create this user was `12345678`. For that reason, the `#authenticate` returns `false`, indicating that the password given was wrong:

```
2.3.3 :007 > u.authenticate('1234567')
 => false
```

(the above code snippet online)

Then we call the `#authenticate` method again with the correct password. In that case, the return value is not `false`, but it is the actual User object again. This makes sure that the password given was correct and the authentication was successful.

Note that `User` model can be created also using an extra attribute called `password_confirmation` too. This is not saved in the database, but it is used to double check the value of the `password` given. Look at the following Ruby statements to create a new `User` using the Rails console:

```
2.3.3 :009 > User.create! email: 'bar@gmail.com', password: '12345678', password_\
confirmation: '123'
   (0.2ms)  BEGIN
  User Exists (0.4ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "bar@gmail.com"], ["LIMIT", 1]]
   (0.1ms)  ROLLBACK
ActiveRecord::RecordInvalid: Validation failed: Password confirmation doesn't mat\
ch Password
        from (irb):9
2.3.3 :010 > User.create! email: 'bar@gmail.com', password: '12345678', password_\
confirmation: '12345678'
   (0.1ms)  BEGIN
  User Exists (0.3ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "bar@gmail.com"], ["LIMIT", 1]]
  SQL (0.2ms)  INSERT INTO "users" ("email", "password_digest", "created_at", "up\
dated_at") VALUES ($1, $2, $3, $4) RETURNING "id"  [["email", "bar@gmail.com"], [\
"password_digest", "$2a$10$dDAw0bl/JMcWdk1DmmX18uM4EyuE2AUt2JcylRj2OVcBWC4z4m5o2"\
], ["created_at", "2017-06-04 06:34:27.884005"], ["updated_at", "2017-06-04 06:34\
:27.884005"]]
   (2.1ms)  COMMIT
 => #<User id: 7, email: "bar@gmail.com", password_digest: "$2a$10$dDAw0bl/JMcWdk\
1DmmX18uM4EyuE2AUt2JcylRj2OVc...", created_at: "2017-06-04 06:34:27", updated_at:\
 "2017-06-04 06:34:27">
2.3.3 :011 >
```

(the above code snippet online)

As you can see above, when we gave `User.create! email: 'bar@gmail.com', password: '12345678', password_confirmation: '123'` the `User` was not created. Whereas when we gave `User.create! email: 'bar@gmail.com', password: '12345678', password_confirmation: '12345678'` the `User` was created.

The `password_confirmation` will have to match the `password` only if the `password_confirmation` is given. In other words, `password_confirmation` is optional and users can be created with `email` and `password` values given only.

Sign Up Form

Our model is ready. Let's now create the sign up form.

Routes and End Points

First we need to create one endpoint for the browser to be able to ask for the sign up form. Then we need to create another endpoint for the browser to be able to post the sign up form data. Let's do that inside our `config/routes.rb` file:

```
1  # File: config/routes.rb
2  #
3  Rails.application.routes.draw do
4    resources :articles
5
6    get '/sign_up', to: 'sign_up#new'
7    post '/sign_up', to: 'sign_up#create'
8
9    root 'welcome#index'
10 end
```

(the above code snippet online)

You can see the new entries on lines 6 and 7. Let's see the routes that these entries generate:

```
1  blog $ bin/rake routes
2  ...
3      sign_up GET    /sign_up(.:format)          sign_up#new
4              POST   /sign_up(.:format)          sign_up#create
5  ...
6  blog $
```

(the above code snippet online)

Nice!

Sign Up - New

Let's implement the `#new` action in a new controller `SignUpController`. Here is the `SignUpController` implementation inside a new file `app/controllers/sign_up_controller.rb`:

```
# File: app/controllers/sign_up_controller.rb
#
class SignUpController < ApplicationController
  def new
    @user = User.new
  end
end
```

(the above code snippet online)

The above is a very simple implementation. Note that we instantiate an empty `User` object. This is the *resource* that this action is manipulating.

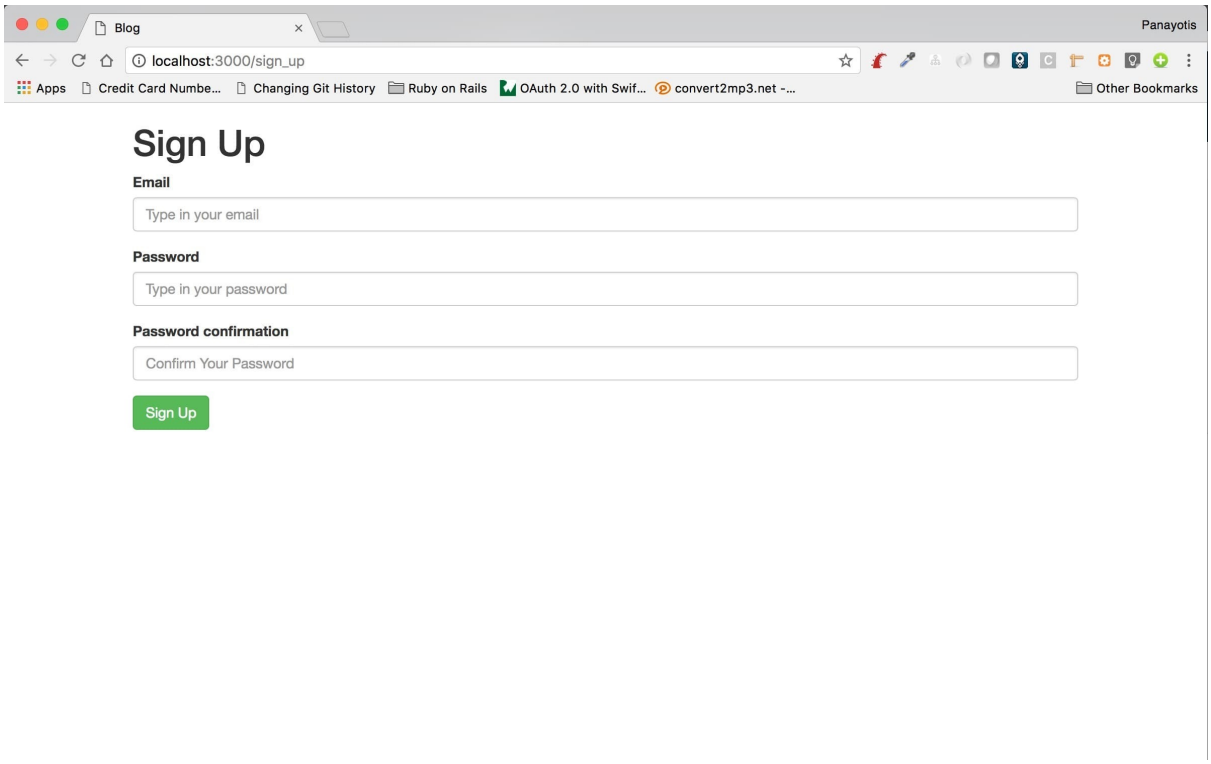
Sign Up Form

Let's now implement that sign up form. It is similar to the form that we use to create a new Article. Create the file `app/views/sign_up/new.html.haml` and put the following content inside. Watch out that you have to create the folder `app/views/sign_up` first. This is the folder that will host the views for the `sign_up` controller.

```
%h1 Sign Up

= form_for @user, url: sign_up_path, method: :post do |f|
  - if @user.errors.any?
    #error-explanation.alert.alert-danger.alert-dismissible{:role => "alert"}
      %button.close{"aria-label" => "Close", "data-dismiss" => "alert", :type => \
"button"}
        %span{"aria-hidden" => "true"} ×
      %h2
        = pluralize(@user.errors.count, "error")
        prohibited User from signing up:
      %ul
        - @user.errors.full_messages.each do |msg|
          %li= msg

  .form-group
    = f.label :email
    = f.email_field :email, placeholder: 'Type in your email', class: 'form-contr\
ol'

  .form-group
    = f.label :password
    = f.password_field :password, placeholder: 'Type in your password', class: 'f\
orm-control'

  .form-group
    = f.label :password_confirmation
    = f.password_field :password_confirmation, placeholder: 'Confirm Your Passwor\
d', class: 'form-control'

  = f.submit 'Sign Up', class: 'btn btn-success'
```

(the above code snippet online)

Now, you are able to fetch the sign up form. Visit the page http://localhost:3000/sign_up. You will see something like this:



Sign Up Form

Sign Up - Create

The form is ready to submit its data. We need to implement the `#create` action in our `SignUpController`. Here it is:

```
# File: app/controllers/sign_up_controller.rb
#
class SignUpController < ApplicationController
  def new
    @user = User.new
  end

  def create
    user = User.new(user_params)
    if user.save
      flash[:success] = 'You have successfully signed up!'
      redirect_to root_url
    else
      @user = user
      render :new
    end
  end

  private
```

```
20
21    def user_params
22      params.require(:user).permit(:email, :password, :password_confirmation)
23    end
24  end
```

(the above code snippet online)

The `#create` method is similar to the implementation of the `#create` method in the `ArticlesController`. There is nothing new for you here. Note that for the time being, if the sign up succeeds, i.e. if the `User` is successfully created, then we redirect to the root path of our Web application.

With all the above in place, the sign up process is finished. You can try that and you will see the `User` being created.

Here is a short video that demonstrates how the Sign Up form works.

Sign Up Demo

Some Links To Ease Navigation

Before we move on, let's create a link in the Welcome page that would allow a user to go to the Sign Up page. Also, from the Sign Up page to return back to the Welcome page.

```
1  %h1 Welcome
2  %div
3    = link_to 'List Of Articles', articles_path
4  %div
5    = link_to 'Create New Article', new_article_path
6  %div
7    = link_to 'Sign Up', sign_up_path
```

(the above code snippet online)

The above is the new Welcome page. The lines 6 and 7 add a new link to the Sign Up page.

Also, add the following at the bottom of your Sign Up view template (`app/views/sign_up/new.html.haml`):

```
1  %div
2    = link_to 'Home', root_path
```

(the above code snippet online)

This will allow a user to go back to `Home` page.

Sign In

For the Sign In feature, we will need the following end points:

1. The end point that will allow the browser to ask for the Sign In page.
2. The end point that will allow the browser to submit the Sign In data.

Routes

Here are the amendments that we need to carry out inside our `config/routes.rb` file in order to support the two endpoints needed for the sign in.

```
1  # File: config/routes.rb
2  #
3  Rails.application.routes.draw do
4    resources :articles
5
6    get '/sign_up', to: 'sign_up#new'
7    post '/sign_up', to: 'sign_up#create'
8
9    get '/sign_in', to: 'sign_in#new'
10   post '/sign_in', to: 'sign_in#create'
11
12   root 'welcome#index'
13 end
```

(the above code snippet online)

The two new lines are line 9 and line 10. What routes do they create?

```
blog $ bin/rake routes
...
     sign_in GET    /sign_in(.:format)          sign_in#new
             POST   /sign_in(.:format)          sign_in#create
...
```

(the above code snippet online)

Nice!

Sign In - New and Create

Let's now create the `SignInController`. Here it is:

```
# File app/controllers/sign_in_controller.rb
#
class SignInController < ApplicationController
  def new
    @sign_in = SignIn.new
  end

  def create
    sign_in = SignIn.new(sign_in_params)
    if sign_in.sign_in
```

```
11        flash[:success] = 'You have successfully sign in!'
12        redirect_to root_url
13      else
14        @sign_in = sign_in
15        render :new
16      end
17    end
18
19    private
20
21    def sign_in_params
22      params.require(:sign_in).permit(:email, :password)
23    end
24  end
```

(the above code snippet online)

The `SignIn` controller follows the same pattern like the controllers that we have already designed so far. However, on line 10, we call `#sign_in` method and not `#save` method. This is because this model is not bound to any database table. So, there is no meaning for a `#save` method. The `#sign_in` method is going to be a custom method implementation in the class `SignIn` that we are going to see next.

`SignIn` Model

The `SignIn` class is going to be a `Model` too. However, this is not an `ApplicationRecord`. The `ApplicationRecord` models are bound to a table in the database. This is only a model without persistent state. The models that they do not have persistent state but they do have attributes and validation rules, we make them be `ActiveModel::Models`. Here is the `SignIn` class implementation:

Let's create the class `SignIn` in the file `app/models/sign_in.rb`:

```
1   # File: app/models/sign_in.rb
2   #
3   class SignIn
4     include ActiveModel::Model
5
6     attr_accessor :email, :password
7
8     validates :email, presence: true
9     validates :password, presence: true
10    validate :authenticate
11
12    def sign_in
13      return false unless valid?
14
```

```
    user
  end

  private

  attr_reader :user

  def authenticate
    return unless email.present? && password.present?

    @user = User.find_by_email(email.downcase).try(:authenticate, password)

    unless @user
      error_message = 'Wrong Credentials!'
      errors[:email] << error_message
      errors[:password] << error_message
    end
  end
end
```

(the above code snippet online)

It does worth explaining some parts of this class:

Not ActiveRecord Model

This model is not an `ActiveRecord` model, but it is an `ActiveModel` model. This is thanks to the `include ActiveModel::Model` module.

Attributes

Since this is not an `ActiveRecord` model, its attributes cannot be derived from the corresponding table, because there is no such binding to a table. Hence, we have to define the attributes explicitly. This is usually done with the `attr_accessor` method:

```
attr_accessor :email, :password
```

(the above code snippet online)

Validations

Although it is not an `ActiveRecord` model, we can still use validations, because it is an `ActiveModel::Model`. That is why we have added the validations:

```
validates :email, presence: true
validates :password, presence: true
validate :authenticate
```

(the above code snippet online)

The new thing here is the `validate :authenticate` statement. The `validate` hooks a custom validation. The custom validation is specified with the name (as symbol) of the private method that implements this validation. Hence, we have the method `authenticate`, later on, that implements the validation.

Action on Model

The action that the controller is calling on the model instantiated is the method `#sign_in`. This will do things as long as the model is valid. Currently, it does not do anything other than returning an authenticated `User` instance.

Custom Validation

The custom validation is implemented with the method

```
def authenticate
  return unless email.present? && password.present?

  @user = User.find_by_email(email.downcase).try(:authenticate, password)

  unless @user
    error_message = 'Wrong Credentials!'
    errors[:email] << error_message
    errors[:password] << error_message
  end
end
```

(the above code snippet online)

This is very simple. If the `email` and `password` are present we are trying to see whether they correspond to valid user with matching password. Note that we are using the `try(...)` Ruby method call that will try to call `:authenticate` with parameter `password` on something that is a `User` or `nil`. If the user is found and stored as `User` instance in the variable `@user`, then we don't have any errors. Otherwise, if `@user` is `nil`, then we use the `error` method on the model to set error messages to specific attributes of the model. Here, we chose to add the same error to all the attributes of the model, since we don't know (and we don't want to reveal) which attribute was really on error (email or password).

Links To Ease Navigation

Let's add a link onto our welcome page that would allow the user to sign in. This is our `app/views/welcome/index.html.haml` page:

```
%h1 Welcome
%div
  = link_to 'List Of Articles', articles_path
%div
  = link_to 'Create New Article', new_article_path
%div
  = link_to 'Sign Up', sign_up_path
%div
  = link_to 'Sign In', sign_in_path
```

(the above code snippet online)

And let's add a link from Sign In page back to Home page. Put this at the end of your `app/views/sign_in/new.html.haml` file:

```
%div
  = link_to 'Home', root_path
```

(the above code snippet online)

Sign In Demo

Now everything should be working as expected. The Sign Up page signs user up and the Sign In page signs user in. This video demonstrates the sign in feature for the user that has been previously signed up (email: `foo@gmail.com` and password: `12345678`):

Sign In Demo

Current User & Sign Out

We have implemented the Sign Up and Sign In functionality. But we don't keep track of the currently signed in user. Also, we don't allow the user to Sign Out so that another user can Sign In.

In order to keep track of the current user and allow for the signing out functionality there are some things that we need to do:

Current User in Session

We will keep track of the current user in the *session*. We will store the current user id in the *session*. Hence, when a user signs in, we will put its `id` into the session.

Here is the new version of the `#create` action of the `SignInController`:

```
def create
  sign_in = SignIn.new(sign_in_params)
  if user = sign_in.sign_in
    session[:user_id] = user.id
    flash[:success] = 'You have successfully sign in!'
    redirect_to root_url
  else
    @sign_in = sign_in
    render :new
  end
end
```

(the above code snippet online)

We take advantage of the return value of the `sign_in.sign_in` call. If this is truthy, then it is the actual user that has signed in. We save the `user.id` into the `session` `Hash` under the key `user_id`.

This means that we can now implement a method that would return the current user. This method is called `current_user` and it is implemented as a `private` method inside the base of all controllers, i.e. inside the `ApplicationController`:

```
# File: app/controllers/application-controller
#
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception

  helper_method :current_user

  private

  def current_user
    User.find_by_id(session[:user_id])
  end
end
```

(the above code snippet online)

You can see that the `current_user` implementation if very simple. It tries to locate the user by `id`. The `id` given as input argument to the `find_by_id` is the id stored in the `sessions` in the key `user_id`. If the user is not signed in, then the `session[:user_id]` will be `nil` and no user will be found.

Note that this method can now be used in any of our controllers, since our controllers derive from `ApplicationController`. Also, note that we have flagged this method as a `helper_method`. This means that it can be used in view templates.

Display Current User

With the ability to have access to the current user we can take advantage of it and display their email on the top right corner. Here is a new version of the `app/views/layouts/application.html.haml`:

```
!!!
%html
  %head
    %meta{:content => "text/html; charset=UTF-8", "http-equiv" => "Content-Type"}/
    %title Blog
    = csrf_meta_tags
    = stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track'\
: 'reload'
    = javascript_include_tag 'application', 'data-turbolinks-track': 'reload'
  %body
    .container
      = render partial: 'layouts/flash'
      = render partial: 'layouts/signed_in_user'
      = yield
```

(the above code snippet online)

There is only one new line, line 12. It renders the partial `signed_in_user`. This partial is the actual file `app/views/layouts/_signed_in_user.html.haml`:

```
- # File: app/views/layouts/_signed_in_user.html.haml
- #
.signed-in-user-container.text-right
 .signed-in-user.text-success
   = current_user.email if current_user
```

(the above code snippet online)

This partial calls the method `current_user` and displays their email, if `current_user` is present.

Sign Out a User

Now that we can display the signed in user, we need to make sure that a user can sign out. We need an endpoint for it. Let's edit the `config/routes.rb` file:

```
# File: config/routes.rb
#
Rails.application.routes.draw do
  resources :articles

  get '/sign_up', to: 'sign_up#new'
  post '/sign_up', to: 'sign_up#create'

  get '/sign_in', to: 'sign_in#new'
  post '/sign_in', to: 'sign_in#create'

  get '/sign_out', to: 'sign_in#destroy'
```

```
13   delete '/sign_out', to: 'sign_in#destroy'
14
15   root 'welcome#index'
16 end
```

(the above code snippet online)

You can see the two new routes, on lines 12 and 13. They both point to the `SignInController` and action `#destroy`. Let's see the actual routes with the `bin/rake routes` command:

```
1 blog $ bin/rake routes
2 ...
3     sign_out GET    /sign_out(.:format)          sign_in#destroy
4              DELETE /sign_out(.:format)          sign_in#destroy
5 ...
6 blog $
```

(the above code snippet online)

Nice! We have implemented two verbs, `GET` and `DELETE`. `DELETE` matches better to the semantics of signing out a user. On the other hand some times it's convenient to sign out a user by just visiting the address `/sign_out` with your browser.

Let's now implement the `#destroy` action inside the `SignInController`. It should be something like this:

```
1 def destroy
2   session.clear
3   redirect_to root_url
4 end
```

(the above code snippet online)

The statement `session.clear` makes sure that whatever is inside the `session` bucket is destroyed. Hence, we don't have any information stored in the `session` that would be bound to a particular user.

Let's put the link to sign out below the signed in user email:

```
1 - # File: app/views/layouts/_signed_in_user.html.haml
2 - #
3 .signed-in-user-container.text-right
4   .signed-in-user.text-success
5     = current_user.email if current_user
6   .sign-out-link
7     = link_to 'Sign Out', sign_out_path, method: :delete
```

(the above code snippet online)

and in the Welcome page:

```
- # File: app/views/welcome/index.html.haml
- #
%h1 Welcome
%div
  = link_to 'List Of Articles', articles_path
%div
  = link_to 'Create New Article', new_article_path
%div
  = link_to 'Sign Up', sign_up_path
%div
  = link_to 'Sign In', sign_in_path
%div
  = link_to 'Sign Out', sign_out_path, method: :delete
```

(the above code snippet online)

1. So, `SignInController#create` adds the `user.id` to the session.
2. `SignInController#destroy` removes the `user.id` from the session.
3. `current_user` checks the `session` for a `user_id` and then tries to locate the User from the database.
4. We display the current user email and a Sign Out link.

The following video shows all the functionality in action:

Sign In and Sign Out Demo

Feature Consistency

Offering `Sign Out` link when no user is signed in, does not make sense. Also, offering `Sign In` link when a user is already in does not make sense either. Other links don't make sense under specific conditions too.

Let's fix that. First, let's make sure that we have consistency in our controllers:

Signing In When Already Signed In

The actions `#new` and `#create` in the `SignInController` do not make sense if you are already signed in.

Here is the implementation of these two actions that now take into consideration that the user might be already signed in:

```ruby
# File: app/controllers/sign_in_controller.rb
#
...
def new
  if current_user
    flash[:warning] = 'Already Signed In!'
    redirect_to root_url
  else
    @sign_in = SignIn.new
  end
end

def create
  if current_user
    flash[:warning] = 'Already Signed In!'
    redirect_to root_url
  else
    sign_in = SignIn.new(sign_in_params)
    if user = sign_in.sign_in
      session[:user_id] = user.id
      flash[:success] = 'You have successfully sign in!'
      redirect_to root_url
    else
      @sign_in = sign_in
      render :new
    end
  end
end
...
```

(the above code snippet online)

Both actions, if the user is already signed in, they redirect to the `/` path and they display a warning flash message.

Try that on your browser:

Signing In When Already Signed In

Signing Out When Not Signed In

Similarly, trying to sign out when you are not signed in, does not make sense either. This is the new version of the `#destroy` action of the `SignInController`:

```
def destroy
  if current_user
    session.clear
  else
    flash[:warning] = 'Not Signed In!'
  end
  redirect_to root_url
end
```

(the above code snippet online)

In the following video, we are trying to sign out, when we are not signed in:

Signing Out When Not Signed In

Signing Up When Already Signed In

For the same reasons, there is no point to sign up when you are already signed in. Let's amend our `SignUpController` accordingly:

```
# File: app/controllers/sign_up_controller.rb
#
class SignUpController < ApplicationController
  def new
    if current_user
      flash[:warning] = 'Already Signed In!'
      redirect_to root_url
    else
      @user = User.new
    end
  end

  def create
    if current_user
      flash[:warning] = 'Already Signed In!'
      redirect_to root_url
    else
      user = User.new(user_params)
      if user.save
        flash[:success] = 'You have successfully signed up!'
        redirect_to root_url
      else
        @user = user
        render :new
      end
    end
  end
```

```

  private

  def user_params
    params.require(:user).permit(:email, :password, :password_confirmation)
  end
end
```

(the above code snippet online)

You can see that we have amended the `#new` action and the `#create` action to check for current_-user.

The following video shows what happens when you try to sign up when you already have signed in:

Signing Up When Already Signed In

Reducing Controller Code Duplication With Filters

If one looks at the code of the `SignUpController`, they will see that there is some code duplication in regards to checking whether the user is already signed in. We can remove this duplication by moving this check into a *before filter*. The filters, the *before* filters can run before every action and they can prevent the action from processing if they `render` or if they `redirect`.

Here is a new version of the `SignUpController` that takes advantage of the `before` filters:

```
# File: app/controllers/sign_up_controller.rb
#
class SignUpController < ApplicationController
  before_action :redirect_if_signed_in

  def new
    @user = User.new
  end

  def create
    user = User.new(user_params)
    if user.save
      flash[:success] = 'You have successfully signed up!'
      redirect_to root_url
    else
      @user = user
      render :new
    end
  end

  private
```

```
22
23    def user_params
24      params.require(:user).permit(:email, :password, :password_confirmation)
25    end
26
27    def redirect_if_signed_in
28      if current_user
29        flash[:warning] = 'Already Signed In!'
30        redirect_to root_url
31      end
32    end
33  end
```

(the above code snippet online)

You can see that on line 4 we have added a `before_action` registration. It registers the method `redirect_if_signed_in`. The private method `redirect_if_signed_in`, lines 27 to 30, has the logic to check whether the user is already signed in. If there is a user signed in it calls the `redirect_to root_url` (line 30). When a `before` filter *renders* or *redirects* then the action will not run.

You can now see that the `#new` and `#create` actions do not have the code duplication that we had earlier.

We can use a similar technique for the `SignInController`. Here is a better version of it:

```
1   # File app/controllers/sign_in_controller.rb
2   #
3   class SignInController < ApplicationController
4     before_action :redirect_if_signed_in, only: [:new, :create]
5
6     def new
7       @sign_in = SignIn.new
8     end
9
10    def create
11      sign_in = SignIn.new(sign_in_params)
12      if user = sign_in.sign_in
13        session[:user_id] = user.id
14        flash[:success] = 'You have successfully sign in!'
15        redirect_to root_url
16      else
17        @sign_in = sign_in
18        render :new
19      end
20    end
21
22    def destroy
```

```
23      if current_user
24        session.clear
25      else
26        flash[:warning] = 'Not Signed In!'
27      end
28      redirect_to root_url
29    end
30
31    private
32
33    def sign_in_params
34      params.require(:sign_in).permit(:email, :password)
35    end
36
37    def redirect_if_signed_in
38      if current_user
39        flash[:warning] = 'Already Signed In!'
40        redirect_to root_url
41      end
42    end
43  end
```

(the above code snippet online)

On line 37 to 42 we define again the method `redirect_if_signed_in` and on line 4 we install the `before` filter. Note that here, we use the `only` and we specify that this applies only for the `new` and the `create` actions, since we do not want this to apply to the `destroy` action.

However, now, we do have the `redirect_if_signed_in` method defined twice. Once in the `SignInController` and another one in the `SignUpController`.

Let's remove the duplication by moving the definition in the `ApplicationController` class.

```
1   # File: app/controllers/application_controller.rb
2   #
3   class ApplicationController < ActionController::Base
4     protect_from_forgery with: :exception
5
6     helper_method :current_user
7
8     private
9
10    def current_user
11      User.find_by_id(session[:user_id])
12    end
13
14    def redirect_if_signed_in
15      if current_user
```

```
16        flash[:warning] = 'Already Signed In!'
17        redirect_to root_url
18      end
19    end
20  end
```

(the above code snippet online)

The lines are 14 to 29. Make sure that you remove the definition of this method from the `SignInController` and from the `SignUpController`.

Authenticated Pages For Articles

Back to our initial problem, when we start with this chapter. We said that

1. We want the article view pages to be visible by anyone. So, they wouldn't require a signed in user.
2. We want the article editing pages to be visible only to signed in users.So, they would require a signed in user.

Black List All Pages

Generally, for security reasons, we initially black list all pages, i.e. we block the access to all pages. Here is how we do that:

Open the `ApplicationController`, which is the superclass of all controllers and make sure you guard all the actions with the requirement for a signed in user. Here is the new version of the `ApplicationController`:

```
1  # File: app/controllers/application_controller.rb
2  #
3  class ApplicationController < ActionController::Base
4    protect_from_forgery with: :exception
5    before_action :authenticate_user
6
7    helper_method :current_user
8
9    private
10
11   def current_user
12     User.find_by_id(session[:user_id])
13   end
14
15   def redirect_if_signed_in
16     if current_user
17       flash[:warning] = 'Already Signed In!'
18       redirect_to root_url
```

```
19      end
20    end
21
22    def authenticate_user
23      redirect_to sign_in_url unless current_user
24    end
25  end
```

(the above code snippet online)

We have added line 5 and lines 22 to 24. We tell Rails that all actions, will have to have an authenticated user, i.e. `current_user` should be returning a truthy value. Otherwise, we will be redirecting user to the Sign In page.

Now, we are sure that whatever action / new page we add in the future, it will not be accessible from the outside world, at least, by default.

White List Welcome, Sign In and Sign Up Pages

Black listing all pages will not even allow the user to access the Sign In Page. Also, user will not be able to access the Welcome or the Sign Up pages either. Let's remove the authentication requirement from these pages:

1. Open the `WelcomeController` and give a statement to skip the filter for authentication of user:

```
1   1. # File: app/controllers/welcome_controller.rb
2   2. #
3   3. class WelcomeController < ApplicationController
4   4.   skip_before_action :authenticate_user
5   5. end
```

   The `skip_before_action` tells Rails that the `authenticate_user` will not be called for all the actions in `WelcomeController`.
2. Open the `SignInController` and make sure that `authenticate_user` is not called whenever it is not appropriate:

```
1    1. # File app/controllers/sign_in_controller.rb
2    2. #
3    3. class SignInController < ApplicationController
4    4.   skip_before_action :authenticate_user
5    5.   before_action :redirect_if_signed_in, only: [:new, :create]
6    6.
7    7.   def new
8    8.     @sign_in = SignIn.new
9    9.   end
10   10.
11   11.   def create
```

```
12  12.     sign_in = SignIn.new(sign_in_params)
13  13.     if user = sign_in.sign_in
14  14.       session[:user_id] = user.id
15  15.       flash[:success] = 'You have successfully sign in!'
16  16.       redirect_to root_url
17  17.     else
18  18.       @sign_in = sign_in
19  19.       render :new
20  20.     end
21  21.   end
22  22.
23  23.   def destroy
24  24.     if current_user
25  25.       session.clear
26  26.     else
27  27.       flash[:warning] = 'Not Signed In!'
28  28.     end
29  29.     redirect_to root_url
30  30.   end
31  31.
32  32.   private
33  33.
34  34.   def sign_in_params
35  35.     params.require(:sign_in).permit(:email, :password)
36  36.   end
37  37. end
```

Line 4, `skip_before_action :authenticate_user` makes sure that we do not redirect user to sign in page.

3. Open the `SignUpController` and add the `skip_before_action` too:

```
1   1. # File: app/controllers/sign_up_controller.rb
2   2. #
3   3. class SignUpController < ApplicationController
4   4.   skip_before_action :authenticate_user
5   5.   before_action :redirect_if_signed_in
6   6.
7   7.   def new
8   8.     @user = User.new
9   9.   end
10  10.
11  11.   def create
12  12.     user = User.new(user_params)
13  13.     if user.save
14  14.       flash[:success] = 'You have successfully signed up!'
15  15.       redirect_to root_url
16  16.     else
```

```
17  17.       @user = user
18  18.       render :new
19  19.     end
20  20.   end
21  21.
22  22.   private
23  23.
24  24.   def user_params
25  25.     params.require(:user).permit(:email, :password, :password_confirmation)
26  26.   end
27  27. end
```

Authenticate the Articles Pages

Now, let's see what we have to do in the `ArticlesController` so that we have the view pages free but the edit pages only for authenticated users. This is the new version of the `ArticlesController`:

```
1  # File: app/controllers/articles_controller.rb
2  #
3  class ArticlesController < ApplicationController
4    skip_before_action :authenticate_user, only: [:index, :show]
5    before_action :find_article, only: [:show, :edit, :update, :destroy]
6
7    def new
8      @article = Article.new
9    end
10
11   def create
12     article = Article.new(article_params)
13     if article.save
14       flash[:success] = 'Article has been created!'
15       redirect_to edit_article_url(article)
16     else
17       @article = article
18       render :new
19     end
20   end
21
22   def index
23     @articles = Article.all
24   end
25
26   def update
27     if @article.update(article_params)
28       flash[:success] = 'Article has been updated!'
```

```
29        redirect_to edit_article_path(@article)
30      else
31        render :edit
32      end
33    end
34
35    def destroy
36      @article.destroy
37      flash[:success] = 'Article has been destroyed!'
38      redirect_to articles_path
39    end
40
41    private
42
43    def article_params
44      params.require(:article).permit(:title, :text)
45    end
46
47    def find_article
48      @article = Article.find(params[:id])
49    end
50  end
```

(the above code snippet online)

We have added the line 4 (after having removed the Basic Authentication line) that skips the authentication for the `index` and `show` action. The authentication remains for the other actions of the `ArticlesController`.

The video below demonstrates the authentication feature that we have added. You can now see how the view pages of articles are accessible by anyone, but the edit pages are only accessible by users that they have signed in.

Articles Public and Private Pages Authentication

Returning Back To Private Page After Sign In

It is very common for the Web application to be redirecting the user to the initially requested protected page after they have signed in. In other words, when a non-signed in user goes to the page http://localhost:3000/articles/new for example, the Web application redirects them to the Sign In page, and then, after user signs in, they are redirected to the originally requested page, i.e. to the page http://localhost:3000/articles/new.

Let's implement this feature.

1. You need to enhance the sign in functionality. When user signs in, we redirect to either the originally requested page or to the `root_url`. Look at the new implementation of the `#create` action of the `SignInController`:

```ruby
def create
  sign_in = SignIn.new(sign_in_params)
  if user = sign_in.sign_in
    session[:user_id] = user.id
    flash[:success] = 'You have successfully sign in!'
    redirect_to params[:return_to] || root_url
  else
    @sign_in = sign_in
    render :new
  end
end
```

The only changes is on line that redirects. It will redirect to whatever the parameter with value `return_to` has or, if this parameter does not exist, to the `root_url`. This means that if the Sign In form submits the `return_to` and it has a value, this is going to be used for the redirect.

2. Hence, we need to open the sign in form and create the hidden field with the value being the parameter with key `return_to`: Open the file `app/views/sign_in/new.html.haml` and make sure it has the following content:

```
  1. %h1 Sign In
  2.
  3. = form_for @sign_in, url: sign_in_path, method: :post do |f|
  4.
  5.   - if @sign_in.errors.any?
  6.     #error-explanation.alert.alert-danger.alert-dismissible{:role => "alert"}
  7.       %button.close{"aria-label" => "Close", "data-dismiss" => "alert", :typ\
e => "button"}
  8.         %span{"aria-hidden" => "true"} ×
  9.       %h2
 10.         = pluralize(@sign_in.errors.count, "error")
 11.         prohibited User from signing in:
 12.       %ul
 13.         - @sign_in.errors.full_messages.each do |msg|
 14.           %li= msg
 15.
 16.   = hidden_field_tag :return_to, params[:return_to]
 17.
 18.   .form-group
 19.     = f.label :email
 20.     = f.email_field :email, placeholder: 'Type in your email', class: 'form-\
control'
 21.
 22.   .form-group
 23.     = f.label :password
 24.     = f.password_field :password, placeholder: 'Type in your password', clas\
s: 'form-control'
```

```
25.
26.   = f.submit 'Sign In', class: 'btn btn-success'
27.
28. %div
29.   = link_to 'Home', root_path
```

It is line 16 that has been added.

3. Now that the form Sign In can post the value of the parameter with key `return_to`, all that it remains is to amend the `authenticate_user` method so that it calls/redirects to the sign in page with `return_to` key value set to the originally requested path. Open the `ApplicationController` class and make sure that it has the following content:

```
1.  # File: app/controllers/application_controller.rb
2.  #
3.  class ApplicationController < ActionController::Base
4.    protect_from_forgery with: :exception
5.    before_action :authenticate_user
6.
7.    helper_method :current_user
8.
9.    private
10.
11.   def current_user
12.     User.find_by_id(session[:user_id])
13.   end
14.
15.   def redirect_if_signed_in
16.     if current_user
17.       flash[:warning] = 'Already Signed In!'
18.       redirect_to root_url
19.     end
20.   end
21.
22.   def authenticate_user
23.     redirect_to sign_in_url(return_to: request.path) unless current_user
24.   end
25. end
```

The only difference is the line 23. Instead of `redirect_to sign_in_url unless current_user`, we do `redirect_to sign_in_url(return_to: request.path) unless current_user`. Hence, we send as parameter the key `return_to` with value the currently requested path that requires authentication to be accessed.

The following video shows how accessing a protected page will first take you to the sign in page and then to the originally requested page:

Return To The Originally Requested Page After Sign In

Hiding Links That Do Not Apply

We have protected our actions from being called under non-valid conditions. For example, the action `#create` on `SignInController` cannot be invoked if the user is already signed in.

But, besides that, we can also hide the links that do not make sense, according to whether the user is signed in or not.

Hide Links on Welcome Page

On the Welcome page we are going to hide the links that should not be displayed according to whether the user is signed in or not. Here is the new version of it:

```
- # File: app/views/welcome/index.html.haml
- #
%h1 Welcome
%div
  = link_to 'List Of Articles', articles_path
- if current_user
  %div
    = link_to 'Create New Article', new_article_path
- unless current_user
  %div
    = link_to 'Sign Up', sign_up_path
  %div
    = link_to 'Sign In', sign_in_path
- if current_user
  %div
    = link_to 'Sign Out', sign_out_path, method: :delete
```

(the above code snippet online)

Do you see how we are using the `current_user` helper method? According to the value it returns we decide which links to display and which ones to not display.

Hide Links On Layout

The application layout displays the `Sign Out` link on the top right. This should not be displayed if user is not signed in. Let's amend the `app/views/layouts/_signed_in_user.html.haml` file by putting the div for the signed in user and the `Sign Out` link into an `if` block:

```
- # File: app/views/layouts/_signed_in_user.html.haml
- #
- if current_user
  .signed-in-user-container.text-right
    .signed-in-user.text-success
      = current_user.email if current_user
    .sign-out-link
      = link_to 'Sign Out', sign_out_path, method: :delete
```

(the above code snippet online)

Hide Links on Articles Index Page

On the page that we display the list of Articles, we are going to hide the links that edit or delete an article if the user is not signed in. Here is the new version of the file:

```
- # File: app/views/articles/index.html.haml
- #
%h1 List Of Articles
%table.table.table-striped
  %thead
    %tr
      %th ID
      %th Title
      %th Text
      %th
  %tbody
    - @articles.each do |article|
      %tr
        %td= article.id
        %td= article.title
        %td= article.text
        %td
          = link_to 'Show', article_path(article)
            
          = link_to 'Edit', edit_article_path(article) if current_user
            
          = link_to 'Delete', article_path(article), method: :delete, class: 'btn\
 btn-danger', data: {confirm: 'Are you sure that you want to delete this Article?\
'} if current_user
- if current_user
  %div
    = link_to 'Create Another Article', new_article_path
%div
  = link_to 'Home', root_path
```

(the above code snippet online)

Look at lines 20, 21 and 23. They all have a condition under which the link is displayed or not.

Hide Links on Article Show Page

Let's hide the links on Article Show page that should only be visible to signed in users.

```
1  - # File: app/views/articles/show.html.haml
2  - #
3  %div
4    %strong ID:
5    = @article.id
6  %div
7    %strong Title:
8    = @article.title
9  %hr/
10 %div
11   %strong Text:
12 %div
13   = @article.text
14 - if current_user
15   %div
16     = link_to 'Edit Article', edit_article_path(@article)
17   %div
18     = link_to 'Create New Article', new_article_path
19 %div
20   = link_to 'List of Articles', articles_path
21 %div
22   = link_to 'Home', root_path
```

(the above code snippet online)

Look at line 14. We have added an `if` statement so that we display the edit and create new links only if user is signed in.

Partial For Errors

Currently, one can identify some repetition in our code that is related to the display of errors in view templates.

Here is the chunk of code that displays the errors on the Article Edit page:

```
- if @article.errors.any?
 #error-explanation.alert.alert-danger.alert-dismissible{:role => "alert"}
   %button.close{"aria-label" => "Close", "data-dismiss" => "alert", :type => "bu\
tton"}
     %span{"aria-hidden" => "true"} ×
   %h2
     = pluralize(@article.errors.count, "error")
     prohibited this article from being saved:
   %ul
     - @article.errors.full_messages.each do |msg|
       %li= msg
```

(the above code snippet online)

The above piece of code is repeated in the Article New page too.

Another page that has similar piece of code is the Sign In Page. Here is its part that deals with errors:

```
- if @sign_in.errors.any?
  #error-explanation.alert.alert-danger.alert-dismissible{:role => "alert"}
    %button.close{"aria-label" => "Close", "data-dismiss" => "alert", :type => "b\
utton"}
      %span{"aria-hidden" => "true"} ×
    %h2
      = pluralize(@sign_in.errors.count, "error")
      prohibited User from signing in:
    %ul
      - @sign_in.errors.full_messages.each do |msg|
        %li= msg
```

(the above code snippet online)

Lastly, another page that has similar code for errors is the Sign Up Page:

```
- if @user.errors.any?
  #error-explanation.alert.alert-danger.alert-dismissible{:role => "alert"}
    %button.close{"aria-label" => "Close", "data-dismiss" => "alert", :type => "b\
utton"}
      %span{"aria-hidden" => "true"} ×
    %h2
      = pluralize(@user.errors.count, "error")
      prohibited User from signing up:
    %ul
      - @user.errors.full_messages.each do |msg|
        %li= msg
```

(the above code snippet online)

Can we create a partial file that will be shared among all these view templates? Yes, we can. Let's create the file `app/views/shared/_errors.html.haml` inside the new folder `app/views/shared`. The file should have the following content:

```
- # File: app/views/shared/_errors.html.haml
- #
- # Local Variables:
- #
- # model {ActiveModel::Mode}. This should be an ActiveModel::Model that responds\
 to +#errors+
- # explanation {String}. The text that will appear next to the number of errors
- #
- if model.errors.any?
  #error-explanation.alert.alert-danger.alert-dismissible{role: 'alert'}
    %button.close{'aria-label': 'Close', 'data-dismiss': 'alert', type: 'button'}
      %span{'aria-hidden': 'true'} ×
    %h2
      = pluralize(model.errors.count, 'error')
      = explanation
    %ul
      - model.errors.full_messages.each do |msg|
        %li= msg
```

(the above code snippet online)

This is the file that has the partial code to display the errors of any model that has errors. And it is an example of a partial that relies on local variables that should be populated by the actual views the partial is going to be invoked from.

Partial With Local Variables

Now, let's replace the occurrences of the error displaying blocks with invocations to render this particular partial.

The `app/views/articles/edit.html.haml` should be like this:

```
1  - # File: app/views/articles/edit.html.haml
2  - #
3  %h1
4    Update Article with ID: #{@article.id}
5  = form_for @article, url: article_path(@article), method: :put do |f|
6
7    = render partial: 'shared/errors', locals: {model: @article, explanation: 'proh\
8  ibited this Article from being saved:'}
9
10   .form-group
11     = f.label :title
12     = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-con\
13 trol'
14   .form-group
15     = f.label :text
16     = f.text_area :text, placeholder: 'Type in the text of the Article', class: '\
17 form-control'
18   %div
19     = f.submit class: 'btn btn-success'
20
21
22 .text-right
23   = link_to 'List of Articles', articles_path
```

(the above code snippet online)

Do you see the call to `render` on line 7? This is how we render a partial. And you can see how we pass real arguments to its local variables.

Similarly, the `app/views/articles/new.html.haml` goes like this:

```
1  - # File: app/views/articles/new.html.haml
2  - #
3  %h1 Create a New Article
4
5  = form_for @article, url: articles_path do |f|
6
7    = render partial: 'shared/errors', locals: {model: @article, explanation: 'proh\
8  ibited this Article from being saved:'}
9
10   .form-group
11     = f.label :title
12     = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-con\
13 trol'
14   .form-group
15     = f.label :text
16     = f.text_area :text, placeholder: 'Type in the text of the Article', class: '\
```

```
form-control'
  %div
    = f.submit class: 'btn btn-success'

.text-right
  = link_to 'List of Articles', articles_path
```

(the above code snippet online)

Next, the `app/views/sign_in/new.html.haml` page:

```
- # File: app/views/sign_in/new.html.haml
- #

%h1 Sign In

= form_for @sign_in, url: sign_in_path, method: :post do |f|

  = render partial: 'shared/errors', locals: {model: @sign_in, explanation: 'proh\
ibited User from signing in:'}

  = hidden_field_tag :return_to, params[:return_to]

  .form-group
    = f.label :email
    = f.email_field :email, placeholder: 'Type in your email', class: 'form-contr\
ol'

  .form-group
    = f.label :password
    = f.password_field :password, placeholder: 'Type in your password', class: 'f\
orm-control'

  = f.submit 'Sign In', class: 'btn btn-success'

%div
  = link_to 'Home', root_path
```

(the above code snippet online)

And finally the `app/views/sign_up/new.html.haml` page:

```
- # File: app/views/sign_up/new.html.haml
- #
%h1 Sign Up

= form_for @user, url: sign_up_path, method: :post do |f|

  = render partial: 'shared/errors', locals: {model: @user, explanation: 'prohibi\
ted User from signing up:'}

  .form-group
    = f.label :email
    = f.email_field :email, placeholder: 'Type in your email', class: 'form-contr\
ol'

  .form-group
    = f.label :password
    = f.password_field :password, placeholder: 'Type in your password', class: 'f\
orm-control'

  .form-group
    = f.label :password_confirmation
    = f.password_field :password_confirmation, placeholder: 'Confirm Your Passwor\
d', class: 'form-control'

  = f.submit 'Sign Up', class: 'btn btn-success'

%div
  = link_to 'Home', root_path
```

(the above code snippet online)

They all now call `shared/errors` partial with the correct local run-time arguments, for `model` and `explanation`.

Form Similar Code

Again, there is still some similar code left in between the files `app/views/articles/edit.html.haml` and `app/views/articles/new.html.haml`:

```
- # File: app/views/articles/edit.html.haml
- #
%h1 Update Article with ID: #{@article.id}

= form_for @article, url: article_path(@article), method: :put do |f|

  = render partial: 'shared/errors', locals: {model: @article, explanation: 'prohibited this Article from being saved:'}

  .form-group
    = f.label :title
    = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-control'
  .form-group
    = f.label :text
    = f.text_area :text, placeholder: 'Type in the text of the Article', class: 'form-control'
  %div
    = f.submit class: 'btn btn-success'

.text-right
  = link_to 'List of Articles', articles_path
```

```
articles/new.html.haml ×
div div
- #
%h1 Create a New Article

= form_for @article, url: articles_path do |f|

  = render partial: 'shared/errors', locals: {model: @article, explanation: 'prohibited this Article from being saved:'}

  .form-group
    = f.label :title
    = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-control'
  .form-group
    = f.label :text
    = f.text_area :text, placeholder: 'Type in the text of the Article', class: 'form-control'
  %div
    = f.submit class: 'btn btn-success'

.text-right
  = link_to 'List of Articles', articles_path
```

Similar Code Between Edit and New

We can definitely extract this into a partial and reduce duplication. Create the file `app/views/articles/_form_fields.html.haml` with the following content:

```
- # File: app/views/articles/_form_fields.html.haml
- #
- # Locals:
- #
- # f: Needs to be a Form Builder
-
= render partial: 'shared/errors', locals: {model: @article, explanation: 'prohib\
ited this Article from being saved:'}

.form-group
  = f.label :title
  = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-contr\
ol'

.form-group
  = f.label :text
  = f.text_area :text, placeholder: 'Type in the text of the Article', class: 'fo\
rm-control'

%div
  = f.submit class: 'btn btn-success'
```

(the above code snippet online)

Then, change your `app/views/articles/edit.html.haml` file to render this partial:

```
- # File: app/views/articles/edit.html.haml
- #
%h1 Update Article with ID: #{@article.id}

= form_for @article, url: article_path(@article), method: :put do |f|

  = render partial: 'articles/form_fields', locals: {f: f}

.text-right
  = link_to 'List of Articles', articles_path
```

(the above code snippet online)

And the `app/views/articles/new.html.haml` file to render the same partial:

```
- # File: app/views/articles/new.html.haml
- #
%h1 Create a New Article

= form_for @article, url: articles_path do |f|

  = render partial: 'articles/form_fields', locals: {f: f}

.text-right
  = link_to 'List of Articles', articles_path
```

(the above code snippet online)

Try to play with your application. You will see that your edit and creation of Articles features work as before.

Form Properties Derive From Model State

We can improve our form code even more but first we need to understand that Rails is clever enough to derive both the `url` and the correct `method` to instantiate the `form` with, based on the actual value that the instance variable `@article` has.

1. If the `@article` is a new, non-persistent instance, i.e. it has not been saved yet in the database, then `form_for` knows that it has to generate a `form` with `action` pointing to `/articles` path and `method` being `post`. This is the case of the `app/views/articles/new.html.haml` view template.

2. If the `@article` is an instance of a persistent Article, i.e. it represents an Article that it will be updated, then `form_for` knows that it has to generate a `form` with the `action` pointing to `/articles/:id` (`:id` being the actual id of the Article being updated) and the `method` being `patch` (which, routing-wise, is handled like the `put`).

Having said that, the next two statements:

```
= form_for @article, url: article_path(@article), method: :put do |f|
```

(the above code snippet online)

and

```
= form_for @article, url: articles_path do |f|
```

(the above code snippet online)

can both be written like this:

```
= form_for @article do |f|
```

(the above code snippet online)

Please, go ahead and do this change in your files `app/views/articles/edit.html.haml` and `app/views/articles/new.html.haml`. You will see that your application behaves exactly the same as before and you can create and update articles without any problem.

Remove Duplication Once More

Now that we have done this improvement above, then the form in file `app/views/articles/edit.html.haml` is exactly the same like the form in the file `app/views/articles/new.html.haml`:

```
- # File: app/views/articles/edit.html.haml
- #
%h1 Update Article with ID: #{@article.id}

= form_for @article do |f|

  = render partial: 'articles/form_fields', locals: {f: f}

.text-right
  = link_to 'List of Articles', articles_path
```

(the above code snippet online)

```
- # File: app/views/articles/new.html.haml
- #
%h1 Create a New Article

= form_for @article do |f|

  = render partial: 'articles/form_fields', locals: {f: f}

.text-right
  = link_to 'List of Articles', articles_path
```

(the above code snippet online)

Let's move the duplication inside the partial `app/views/articles/_form_fields.html.haml` and, actually, rename that file being `app/views/articles/_form.html.haml` since will now be including the whole form. Doing this, it will not require to pass the `f` form builder as a local parameter too.

Here is the new version of the partial which is now called `app/views/articles/_form.html.haml`:

```
- # File: app/views/articles/_form.html.haml
- #
= form_for @article do |f|
  = render partial: 'shared/errors', locals: {model: @article, explanation: 'proh\
ibited this Article from being saved:'}

  .form-group
    = f.label :title
    = f.text_field :title, placeholder: 'Give Title of Article', class: 'form-con\
trol'

  .form-group
    = f.label :text
    = f.text_area :text, placeholder: 'Type in the text of the Article', class: '\
form-control'

  %div
    = f.submit class: 'btn btn-success'
```

(the above code snippet online)

And the new versions of the files `app/views/articles/edit.html.haml` and `app/views/articles/new.html.haml`

```
- # File: app/views/articles/edit.html.haml
- #
%h1 Update Article with ID: #{@article.id}

= render partial: 'articles/form'

.text-right
  = link_to 'List of Articles', articles_path
```

(the above code snippet online)

```
- # File: app/views/articles/new.html.haml
- #
%h1 Create a New Article

= render partial: 'articles/form'

.text-right
  = link_to 'List of Articles', articles_path
```

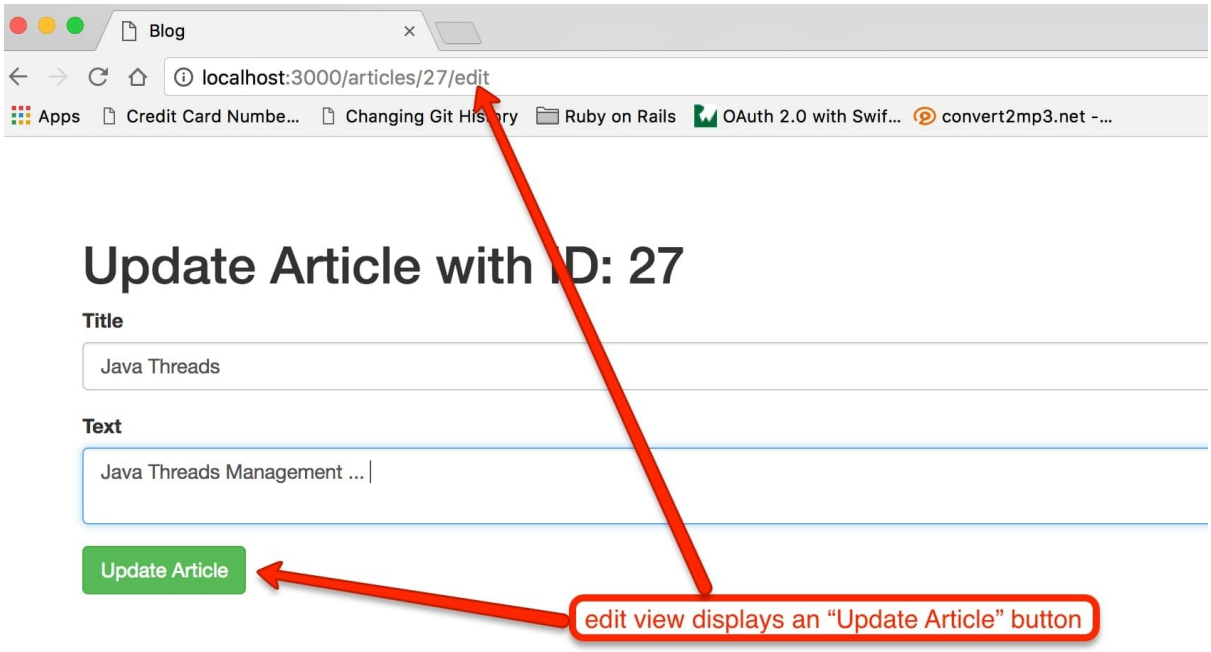
(the above code snippet online)

Note that the cool think about using the same form code for both new (create) and edit (update) functionality is that the `= f.submit class: 'btn btn-success'` call that exists to create the Call To Action button, it creates the correct value / text to display, depending on whether the form is for the creation or the update feature:

Here is the button when we create a new article:

CTA When We Create New Article

Here is the button when we update an existing article:



CTA When We Update Existing Article

Again, `= f.submit` is clever enough to use the correct value for the button label.

Closing Note

That was a long journey to Rails development that touched, mainly, Authentication, but also other parts of your Web application like partials, forms and session.

In the next chapter we are going to talk about sending emails out.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

Task A

All the work that is described in the chapter content needs to be accomplished and pushed to your Github account. Also, it needs to be public using Heroku.

Task B

Continue the development on the e-shop you started in the previous chapter. All the things that you have learned for the session-based authentication needs to be applied. For example:

1. You need to create a User model.
2. You need to implement the Sign Up feature.
3. You need to implement the Sign In feature.
4. You need to make sure that you have links to ease the navigation in the e-shop.
5. You need to display the email of the user currently logged in.
6. You need to implement the Sign Out feature.
7. You need to make sure that actions in controllers are not called if not applicable.
8. You need to reduce the code duplication both at the view level and at the controller level.
9. Product view pages should be public. Whereas Product creation/editing/deletion pages should require authentication.

Make sure that you commit your code to your Github account and that you deploy it to your production server in Heroku too.

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

4 - Sending out Emails

Summary

Sending Out Emails

Sending out emails to users is one of the most interesting feature of a Web application. It enhances the experience of the User with our online business. Not only that, sometimes it is absolutely necessary to communicate with the user, for example when we want them to confirm their email address. This chapter will teach you how Ruby on Rails integrates with an SMTP server and what tools you can use to organize the content and delivery of your emails.

Learning Goals

1. Learn how you can ask your users to confirm their email address.
2. Learn about the dummy SMTP servers, like Mailtrap.
3. Learn how you can integrate your development environment with Mailtrap.
4. Learn how you can send out a test email.
5. Learn how to add a new column onto a table and migrate both schema and data.
6. Learn about some of the ActiveRecord callbacks and how they run code into a transaction.
7. Learn how to use basic `ActionMailer::Base` functionality to send out HTML emails.
8. Learn how to URL encode query params.
9. Learn how to define routes that expect query params too.
10. Learn how to use `ActiveRecord` dynamic finder methods.
11. Learn about custom Mailers and how they work more like Controllers.
12. Learn where the customer Mailers live.

13. Learn how you can specify the default email attributes.
14. Learn how mails use layouts.
15. Learn about the mailer views.
16. Learn how you can support both HTML and plain TEXT emails.
17. Learn how you can have access from the mailer views to the instance variables of the mailer actions.
18. Learn how to create full URLs, with protocol and host included, inside your emails.
19. Learn how to define a schema migration to add an index in one of the database tables.
20. Learn how to be careful when running standard SQL commands.
21. Learn how to integrate with an SMTP server for your production environment.

Introduction

We will enhance our `blog` application to send emails to our users. In fact:

We want the users to confirm their email address when they sign up.

1. We will do that by sending an email to the user with a link to confirm their email address.
2. This email will be sent out when the user signs up.
3. Until the user confirms their email, we will not be able to send any other email to the user, and we will be considering their email unconfirmed.
4. Also, user will be able to request for a new email delivery with a new link to confirm their email address.

Sending Out Emails While In Development

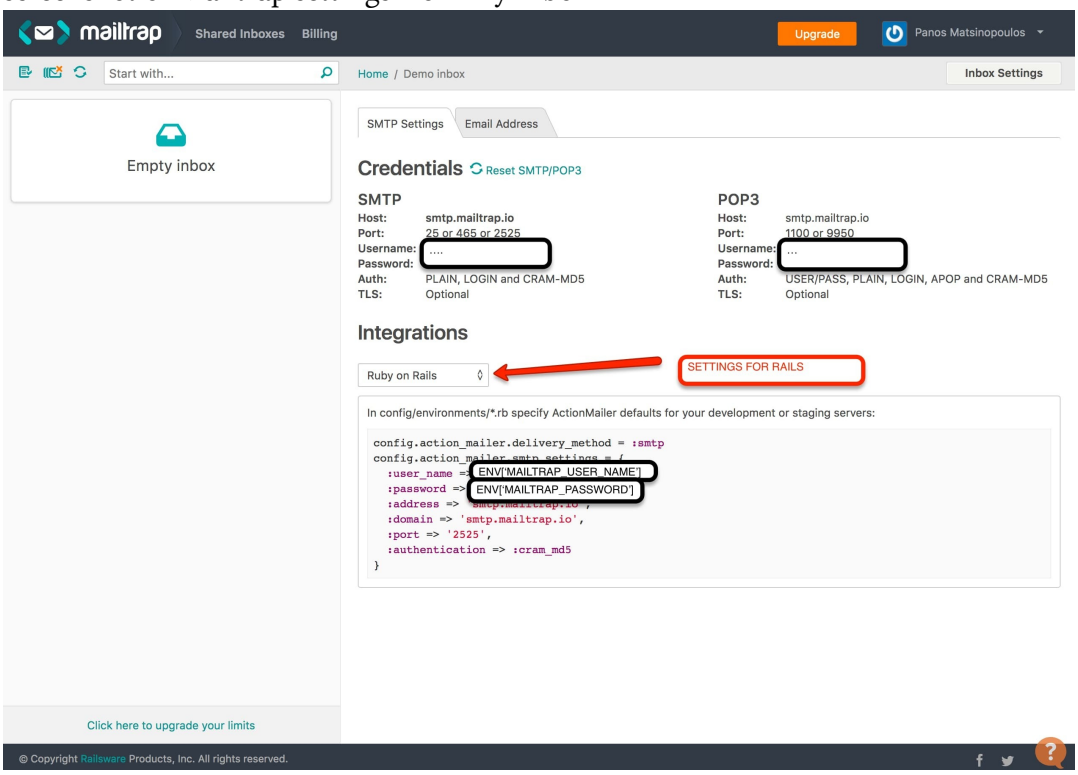
While you will be doing your development, you wouldn't like to have real emails being sent out to real users. There are various ways you can use to prevent you from doing this, but one easy way is to actually integrate with a dummy SMTP (Simple Mail Transfer Protocol) server. Normally, an SMTP server accepts an email out request and makes sure to forward the email out to the destination email box. A dummy SMTP server, on the other hand, accepts the email out request, but stores the email internally and does not send it out. The dummy SMTP server gives you an interface for you to inspect the email content so that you know what would have been sent out if this were a real SMTP server.

Such a dummy SMTP server is Mailtrap.

Integrating With Mailtrap

In order to integrate Mailtrap to your development environment, then you need to first create an account with them. After you do that, you need to set your Mailtrap credentials inside your `config/environments/development.rb` file.

Take your credentials from the inbox settings on your Mailtrap account. Here is a screenshot of Mailtrap settings from my inbox:



Here it is how you should set them:

```
# File: config/environments/development.rb
#
Rails.application.configure do
  # Settings specified here will take precedence over those in config/application\
.rb.

  # In the development environment your application's code is reloaded on
  # every request. This slows down response time but is perfect for development
  # since you don't have to restart the web server when you make code changes.
  config.cache_classes = false

  # Do not eager load code on boot.
  config.eager_load = false

  # Show full error reports.
  config.consider_all_requests_local = true

  # Enable/disable caching. By default caching is disabled.
  if Rails.root.join('tmp/caching-dev.txt').exist?
    config.action_controller.perform_caching = true

```

```
22      config.cache_store = :memory_store
23      config.public_file_server.headers = {
24        'Cache-Control' => "public, max-age=#{2.days.seconds.to_i}"
25      }
26    else
27      config.action_controller.perform_caching = false
28
29      config.cache_store = :null_store
30    end
31
32    # Don't care if the mailer can't send.
33    config.action_mailer.raise_delivery_errors = true
34    config.action_mailer.perform_caching = false
35    config.action_mailer.delivery_method = :smtp
36    config.action_mailer.smtp_settings = {
37      user_name: ENV['MAILTRAP_USER_NAME'],
38      password: ENV['MAILTRAP_PASSWORD'],
39      address: 'smtp.mailtrap.io',
40      domain: 'smtp.mailtrap.io',
41      port: '2525',
42      authentication: :cram_md5
43    }
44
45    # Print deprecation notices to the Rails logger.
46    config.active_support.deprecation = :log
47
48    # Raise an error on page load if there are pending migrations.
49    config.active_record.migration_error = :page_load
50
51    # Debug mode disables concatenation and preprocessing of assets.
52    # This option may cause significant delays in view rendering with a large
53    # number of complex assets.
54    config.assets.debug = true
55
56    # Suppress logger output for asset requests.
57    config.assets.quiet = true
58
59    # Raises error for missing translations
60    # config.action_view.raise_on_missing_translations = true
61
62    # Use an evented file watcher to asynchronously detect changes in source code,
63    # routes, locales, etc. This feature depends on the listen gem.
64    config.file_watcher = ActiveSupport::EventedFileUpdateChecker
65  end
```

(the above code snippet online)

Look at lines 32 to 42. This is how Mailtrap recommends that we set our `config/environments/development.rb`. These lines configure the SMTP server that our Web application will be connected to in order to be sending out emails.

Note that we do not hard code the `:user_name` and `:password` values. They are going to be stored inside the `.env` file and they are being retrieved here dynamically (since we are using the `dotenv-rails` gem).

> **Important:** Don't write your credentials inside the `config/environments/development.rb` file. You don't want these to be checked in into your Git repository.

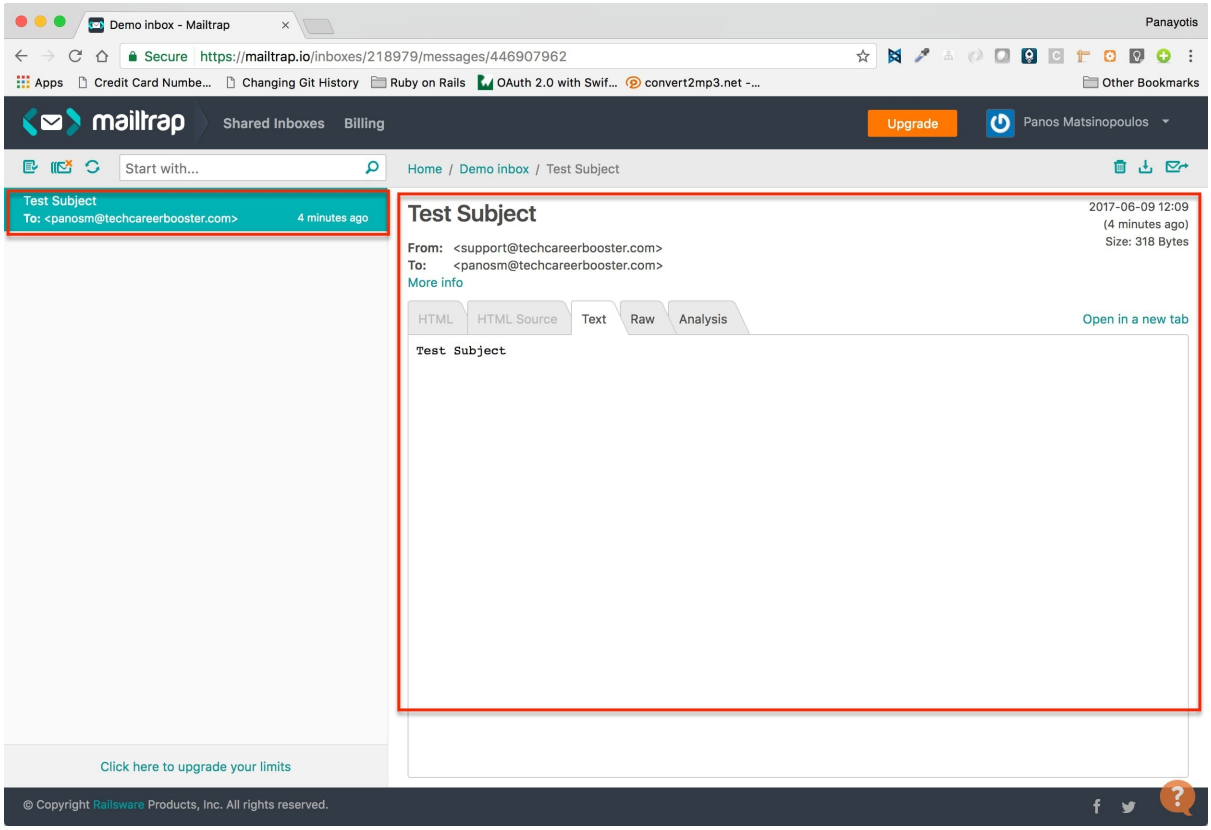
Test Email

Now your integration with Mailtrap is ready. Let's try to send out a test email. Open your rails console:

```
blog $ bin/rails console
2.3.3 :001> ActionMailer::Base.mail(to: 'panosm@techcareerbooster.com', from: 'su\
pport@techcareerbooster.com', subject: 'Test Subject', body: 'Test Subject').deli\
ver
ActionMailer::Base#mail: processed outbound mail in 117.5ms
Sent mail to panosm@techcareerbooster.com (2934.8ms)
Date: Fri, 09 Jun 2017 15:09:24 +0300
From: support@techcareerbooster.com
To: panosm@techcareerbooster.com
Message-ID: <593a8ff45ab31_88f13fd2ed83f9c8223e7@Panayotiss-MacBook-Pro.local.mai\
l>
Subject: Test Subject
Mime-Version: 1.0
Content-Type: text/plain;
 charset=UTF-8
Content-Transfer-Encoding: 7bit

Test Subject
 => #<Mail::Message:70175218012900, Multipart: false, Headers: <Date: Fri, 09 Jun\
 2017 15:09:24 +0300>, <From: support@techcareerbooster.com>, <To: panosm@techcar\
eerbooster.com>, <Message-ID: <593a8ff45ab31_88f13fd2ed83f9c8223e7@Panayotiss-Mac\
Book-Pro.local.mail>>, <Subject: Test Subject>, <Mime-Version: 1.0>, <Content-Typ\
e: text/plain>, <Content-Transfer-Encoding: 7bit>>
2.3.3 :002 >
```

(the above code snippet online)

As you can see above, we have used the `ActionMailer::Base.mail(...)` to construct a mail object and then call the method `deliver` on that to send out the actual email. This mail has been finally caught by Mailtrap, our mailcatching proxy. Here is the screenshot from the email appearing in Mailtrap dashboard:



Test Message Appearing on Mailtrap

Don't worry about the fact that the email you are looking at is only a *text* version of your email. This is because you actually sent out only a text version of your email. You will learn how to send out HTML emails in a while.

We have set up our email integration for the development environment. We will have to set up our email integration for the production environment too, but this is something that we will do at the end of this chapter.

Email is Pending Confirmation

Let's now implement the feature for email confirmation. Users, when they sign up, they need to confirm their email address.

1. When a user signs up we will flag their email address as unconfirmed.
2. We will send them out an email to confirm their email address. The email will have a link that the user should click on.
3. When the user clicks on the email address the link would take the user to the welcome page with a flash message that their email address has been confirmed.

Amend `users` Table

Let's amend the `users` table to have a column with name `email_confirmation_token`. This will be a string and will randomly take a value when the User signs up.

The following is a schema migration generation command that will generate the file for us to specify the changes on `users` table:

```
blog $ bin/rails generate migration add_email_confirmation_token_to_users
invoke  active_record
create    db/migrate/20170610052034_add_email_confirmation_token_to_users.rb
blog $
```

(the above code snippet online)

We now edit the file `db/migrate/20170610052034_add_token_to_users.rb` and make its content as follows:

```
1  # File: db/migrate/20170610052034_add_email_confirmation_token_to_users.rb
2  #
3  require 'securerandom'
4
5  class AddEmailConfirmationTokenToUsers < ActiveRecord::Migration[5.1]
6    def change
7      add_column :users, :email_confirmation_token, :string
8
9      # We need to run a data migration for existing Users
10     User.reset_column_information
11
12     User.find_each do |user|
13       user.update_column(:email_confirmation_token, SecureRandom.urlsafe_base64)
14     end
15   end
16 end
```

(the above code snippet online)

On line 7 we use the `add_column` method call that allows us to add a column to existing table. Then on line 10, we call `User.reset_column_information`. This is necessary for the Rails process that runs the migration to reload the information in regards to the columns that correspond to the `User` model, since we have added a new column. If we don't do that, then line 13, that uses the new column will fail.

The idea of the above schema migration is to first add the new column and then to do a data migration for the existing users in our database. For all the users, it will set a random value to their column `email_confirmation_token`.

Let's now run the schema migration:

```
blog $ bin/rake db:migrate
== 20170610052034 AddEmailConfirmationTokenToUsers: migrating =================
-- add_column(:users, :email_confirmation_token, :string)
   -> 0.0026s
== 20170610052034 AddEmailConfirmationTokenToUsers: migrated (0.0391s) ========

blog $
```

(the above code snippet online)

The migration has run successfully. The following `psql` interaction can be used to confirm the changes on the database level:

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \connect blog_development
You are now connected to database "blog_development" as user "panayotismatsinopou\
los".
blog_development=# \d+ users
                                                                  Table "public.use\
rs"
          Column          |            Type             |                     Mod\
ifiers                      | Storage  | Stats target | Description
--------------------------+-----------------------------+------------------------\
----------------------------+----------+--------------+-------------
 id                       | bigint                      | not null default nextva\
l('users_id_seq'::regclass) | plain    |              |
 email                    | character varying           | not null               \
                            | extended |              |
 password_digest          | character varying           | not null               \
                            | extended |              |
 created_at               | timestamp without time zone | not null               \
                            | plain    |              |
 updated_at               | timestamp without time zone | not null               \
                            | plain    |              |
 email_confirmation_token | character varying           |                        \
                            | extended |              |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)

blog_development=# select * from users;
 id |     email     |                      password_digest                      \
  |         created_at         |         updated_at         | email_confirmation_\
token
----+---------------+-----------------------------------------------------------\
```

```
--+----------------------------+----------------------------+--------------------\
------
  8 | foo@gmail.com | $2a$10$WuL5ic/qRTL2.AK9vAIDxuoHZAf7V/dFdlLzFpgXdnwc1GhtrJj8\
i | 2017-06-04 07:25:59.681835 | 2017-06-04 07:25:59.681835 | 3sRJUZvDS0wNMNY2pfb\
U_A
  9 | bar@gmail.com | $2a$10$g/4AILw8UXB1s8G01M8P1efgzs0iZfNfw.BxdHW2HNIxMhgkgqrr\
2 | 2017-06-04 13:51:46.423347 | 2017-06-04 13:51:46.423347 | dX84eWeLcTyF9-2f6ZM\
01Q
(2 rows)

blog_development=# \q
blog $
```

(the above code snippet online)

You can see that the table `users` now has a new column named `email_confirmation_token` and all the users have a value that has been randomly generated when we ran the schema migration.

`User` Model Changes

We will enhance our `User` model to tell us whether the user has confirmed email address or not. Here is how we will do that:

```ruby
# File app/models/user.rb
#
class User < ApplicationRecord
  validates :email, presence: true, uniqueness: { case_sensitive: false }
  has_secure_password

  def email_confirmed?
    !email_confirmation_token.present?
  end
end
```

(the above code snippet online)

We have added the method `email_confirmed?` This returns `true` when the `email_confirmation_token` is not present, i.e. when it does not exist, and returns `false` when the `email_confirmation_token` is present, i.e. when it exists.

`email_confirmation_token.present?`	Email Confirmed?
`false`	`true`
`true`	`false`

In other words:

1. We want the `User` to be created, signed-up, with a random `email_confirmation_token`.

Hence, `email_confirmed?` will be returning `false`.
2. When the `User` will confirm their email address, this token will be deleted. Hence, `email_confirmed?` will be returning `true`.

But, how can we make sure that whenever we create a `User`, this `User` has a random `email_confirmation_token` attached to it?

Shall we amend the `#create` action in the `SignUpController` to make it attach a token after the user is created? That would have been a solution, but Rails offers a better one that makes our life much easier. It is called ActiveRecord Callbacks. We will talk more about those in a later chapter, but until then, let's use one in our `User` model:

```
1  # File app/models/user.rb
2  #
3  class User < ApplicationRecord
4    validates :email, presence: true, uniqueness: { case_sensitive: false }
5    has_secure_password
6
7    before_create :generate_email_confirmation_token
8
9    def email_confirmed?
10     !email_confirmation_token.present?
11   end
12
13   private
14
15   def generate_email_confirmation_token
16     self.email_confirmation_token = SecureRandom.urlsafe_base64
17   end
18 end
```

(the above code snippet online)

Line 7, `before_create :generate_email_confirmation_token` attaches the method code from the method `generate_email_confirmation_token` at the hook that is invoked exactly before the `User` is created in the database. If you see the implementation of this method, you will see that it creates a random string as an `email_confirmation_token`.

This is a very useful tool for the following reasons:

1. There is no way you can create a `User` without an `email_confirmation_token`. If we put this logic in the `SignUpController`, then that leaves a door open for creation of `Users` without `email_confirmation_token`, e.g. using the `rails console`.
2. Creation of the `User` and the assignment of the `email_confirmation_token` is done in one transaction. So, you don't risk to have created a `User` without them having an `email_confirmation_token`.

Display Unconfirmed Flag

Let's enhance our layout to display an unconfirmed email flag next to the email of the signed in user:

```
1  - # File: app/views/layouts/_signed_in_user.html.haml
2  - #
3  - if current_user
4    .signed-in-user-container.text-right
5      .signed-in-user.text-success
6        = current_user.email
7        - unless current_user.email_confirmed?
8          %span.text-danger
9            = ' (Unconfirmed) '
10     .sign-out-link
11       = link_to 'Sign Out', sign_out_path, method: :delete
```

(the above code snippet online)

The above is the `app/views/layouts/_signed_in_user.html.haml` that has been amended (lines 7 to 9) to display the word (Unconfirmed) if the currently signed in user has an email that has not been confirmed.

Watch the following video displaying this flag for the existing users, but also for a new sign up. Also, it demonstrates how the new User is really signed up with a new email confirmation token.

Unconfirmed Email and Email Confirmation Token

Sending Email Out

We have dealt with creating the email confirmation token for each User. This happens when the sign up takes place, in fact, when the User is created. But how can we hook the delivery of an email to the User with the link to click on in order to confirm their email?

In order to do that, we may want to enhance the `SignUpController` to send out an email to the user that has just signed up.

> *Information:* Why don't we just create another ActiveRecord Callback that would send the email out? For example, ActiveRecord has a callback called `after_create` which could be used for that reason. However, the question here is: *Do we really want to send an email out every time a User is created?* We may want to run a bulk import script to import many Users from another database. If that was the case, would we want to send them an email out? This *send an email out to ask for a confirmation* seems to be more like an application flow responsibility, rather than a domain model responsibility. Having done these thoughts, we believe that it is better if this was handled at the controller layer, which is the layer that deals with application flow and logic.

Here is the new version of our `SignUpController`:

```
1  # File: app/controllers/sign_up_controller.rb
2  #
3  class SignUpController < ApplicationController
4    skip_before_action :authenticate_user
5    before_action :redirect_if_signed_in
6
7    def new
8      @user = User.new
9    end
10
11   def create
12     user = User.new(user_params)
13     if user.save
14       send_email_to_confirm_their_email_address(user)
15       flash[:success] = 'You have successfully signed up! Please, confirm your em\
16 ail address. We just sent you out an email with a link to do so!'
17       redirect_to root_url
18     else
19       @user = user
20       render :new
21     end
22   end
23
24   private
25
26   def user_params
27     params.require(:user).permit(:email, :password, :password_confirmation)
28   end
29
30   def send_email_to_confirm_their_email_address(user)
31     ActionMailer::Base.mail(
32       to: user.email,
33       from: 'support@techcareerbooste.com',
34       subject: 'TCB Blog: Please, Confirm Your Email Address',
35       body: "Please, click <a href='http://localhost:3000/confirm_email_address?e\
36 mail=#{CGI.escape(user.email)}&email_confirmation_token=#{user.email_confirmation\
37 _token}'>here to confirm your email address</a>",
38       content_type: 'text/html'
39     ).deliver
40   end
41 end
```

(the above code snippet online)

Look at lines 14, 15 and then on lines 29 to 37. I guess that everything is easily explained. After we sign up a user, we send an email out and we then redirect to the `root_url`. The `flash` message has been adapted in order to prompt the user to check their email out.

The method `send_email_to_confirm_their_email_address` is very simple to understand. It uses the `ActionMailer::Base.mail(...).deliver` method to send out an email. Note that the `body` now has content that includes the link that the recipient would have to click on. Because it includes HTML content, we had to tell `ActionMailer::Base` that `content_type:` has the value `text/html`. Otherwise, it would have been treated as plain text.

The link that it is included in the email body is of the format: `http://localhost:3000/confirm_email_address?email=<email>&email_confirmation_token=<email_confirmation_token>`. Note also that the `email` is being URL encoded with the help of the `CGI.escape(...)` method. The `email_confirmation_token` does not have to be sent to `CGI.escape(...)` because it is already URL safe, since it is generated with the call to `SecureRandom.urlsafe_base64`.

> **Important:** It is not recommended to use `ActionMailer::Base` to send your emails out. Later on in this chapter you will see what is the proper way to use Rails tools to send out emails.

> **Important:** Usually, we do not ask the Web application to deliver the email within the HTTP request processing cycle, i.e. synchronously. In other words, we do not call `#deliver`. We use a technique to do it asynchronously using background tasks. We will see this technique in a later chapter.

Obviously, if the user clicks on this link we will have a failure because, currently, we have not defined any endpoint to accept such requests. We will do that after we first demonstrate what we have done so far with the following video. In the following video, we sign up a User and then we get a look into our Mailtrap inbox to see that the email for confirmation has been delivered as expected.

Unconfirmed Email and Email Confirmation Token

EndPoint To Accept Confirmation

Ok. Cool! If the user clicks on the link inside the email that they receive, they will get an error.

Let's now amend our `config/routes.rb` file in order to add a new endpoint to accept `GET` requests for email confirmation.

```
# File: config/routes.rb
#
Rails.application.routes.draw do
  resources :articles

  get '/sign_up', to: 'sign_up#new'
  post '/sign_up', to: 'sign_up#create'
  get '/confirm_email_address', to: 'sign_up#confirm_email_address' # ?email=<ema\
il>&email_confirmation_token=<email_confirmation_token>

  get '/sign_in', to: 'sign_in#new'
```

```
12   post '/sign_in', to: 'sign_in#create'
13
14   get '/sign_out', to: 'sign_in#destroy'
15   delete '/sign_out', to: 'sign_in#destroy'
16
17   root 'welcome#index'
18 end
```

(the above code snippet online)

Do you see line 8? This is the new line that we have added. Note that the path specification does not include the `email` and the `email_confirmation_token`. These values are going to be sent over using query params and hence they are not specified in the path entry in routes. However, I have added a comment at the end of the line explaining which are the expected query params this route is going to deal with.

Let's confirm the actual routes that the above file builds:

```
1 blog $ bin/rake routes
2 ...
3 confirm_email_address GET    /confirm_email_address(.:format) sign_up#confirm_ema\
4 il_address
5 ...
6 blog $
```

(the above code snippet online)

Amend `SignUp` Controller

With the above routes in place, we need to add a new action inside `SignUpController`. The action needs to be `confirm_email_address`. Here is the implementation of it:

```
1 # File: app/controllers/sign_up_controller.rb
2 #
3 class SignUpController < ApplicationController
4   skip_before_action :authenticate_user
5   before_action :redirect_if_signed_in, except: [:confirm_email_address]
6
7   def new
8     @user = User.new
9   end
10
11  def create
12    user = User.new(user_params)
13    if user.save
14      send_email_to_confirm_their_email_address(user)
15      flash[:success] = 'You have successfully signed up! Please, confirm your em\
```

```
16 ail address. We just sent you out an email with a link to do so!'
17       redirect_to root_url
18     else
19       @user = user
20       render :new
21     end
22   end
23
24   def confirm_email_address
25     user = User.find_by_email_and_email_confirmation_token!(params[:email], param\
26 s[:email_confirmation_token])
27     if user.confirm_email_address
28       flash[:success] = 'You have successfully confirmed your email address!'
29     else
30       flash[:error] = 'Something went wrong and we cannot confirm your email addr\
31 ess!'
32     end
33     redirect_to root_url
34   end
35
36   private
37
38   def user_params
39     params.require(:user).permit(:email, :password, :password_confirmation)
40   end
41
42   def send_email_to_confirm_their_email_address(user)
43     ActionMailer::Base.mail(
44       to: user.email,
45       from: 'support@techcareerbooste.com',
46       subject: 'TCB Blog: Please, Confirm Your Email Address',
47       body: "Please, click <a href='http://localhost:3000/confirm_email_address?e\
48 mail=#{CGI.escape(user.email)}&email_confirmation_token=#{user.email_confirmation\
49 _token}'>here to confirm your email address</a>",
50       content_type: 'text/html'
51     ).deliver
52   end
53 end
```

(the above code snippet online)

Watch for the line 5. We have excluded the new method `confirm_email_address` from being redirected if the user is already signed in. We don't want that. We want the action to take place even if the user is already signed in.

See the lines 23 to 31 which are the implementation of the email confirmation. First, on line 24, we are trying to locate the User using the two parameters, i.e. the `email` and the

`email_confirmation_token`. The `#find_by_x_y` is a method that is dynamically available to us by `ActiveRecord`. `x` and `y` being any of the model attributes. The `!` at the end makes sure that if the `User` is not located an exception is raised. The exception is `ActiveRecord::RecordNotFound` which causes the Rails server to return back a response with status code `404 Not Found`. We will elaborate on that later on.

Next, on line 25, we call `User#confirm_email_address` to ask the domain model to render the email address confirmed. We have not implemented this method in the model yet. We are going to do that next. Other than that, the `SignUpController#confirm_email_address` is pretty straight forward.

User#confirm_email_address

Here is the implementation of the `User#confirm_email_address`:

```
1  # File app/models/user.rb
2  #
3  class User < ApplicationRecord
4    validates :email, presence: true, uniqueness: { case_sensitive: false }
5    has_secure_password
6
7    before_create :generate_email_confirmation_token
8
9    def email_confirmed?
10     !email_confirmation_token.present?
11   end
12
13   def confirm_email_address
14     self.email_confirmation_token = nil
15     save
16   end
17
18   private
19
20   def generate_email_confirmation_token
21     self.email_confirmation_token = SecureRandom.urlsafe_base64
22   end
23 end
```

(the above code snippet online)

Look at lines 13 to 16. The implementation of the `User#confirm_email_address` is very simple. We just set the `email_confirmation_token` to `nil` and we save.

Now the flow of confirming the email address has been finished. You can try that on your browser. Here is a full demo of that on our own browser:

Full Sign Up Demo With Email Confirmation

Sending Emails Out - The Recommended Way

We have used the `ActionMailer::Base.mail(...).deliver` method to send the email out to the User that has just signed up. However, this is not the recommended way to be sending emails out. We should be using custom Mailers and Mailer Views.

Mailers work more or less like Controllers do. They are custom classes that derive from `ActionMailer::Base` and they have public methods which are called actions. The actions then are matched to view templates, usually in two versions, an HTML version and a TEXT version in order to support email clients that only deal with TEXT-content emails.

Let's see how we should implement our feature in order to use this recommended technique:

Change `SignUpController`

First, we will change the `SignUpController` to be using the Mailer. Here is the new implementation of the method `#send_email_to_confirm_their_email_address`:

```
...
def send_email_to_confirm_their_email_address(user)
  UserMailer.ask_for_email_confirmation(user.id).deliver
end
...
```

(the above code snippet online)

The implementation is now using the `UserMailer` class and calls its action `ask_for_email_confirmation` passing as argument the `id` of the `user` instance. Then it calls the `#deliver` method as before.

`UserMailer`

The mailers live inside the `app/mailers` folder. Let's create the class `UserMailer` in the file `app/mailers/user_mailer.rb` as follows:

```
# File: app/mailers/user_mailer.rb
#
class UserMailer < ApplicationMailer
  def ask_for_email_confirmation(user_id)
    @user = User.find(user_id)
    mail(
      to: @user.email,
      subject: 'TCB Blog: Please, Confirm Your Email Address'
    )
  end
end
```

(the above code snippet online)

You can see that the action implementation is very simple.

It first finds the user and assigns the instance to `@user`. Then it calls the method `mail` which will return a Mail object. The `mail` command, in this particular case, specifies the recipient and the subject. The `from` is specified in the `ApplicationMailer` class, which is the superclass of the custom `UserMailer`:

```
1 # File: app/mailers/application_mailer.rb
2 #
3 class ApplicationMailer < ActionMailer::Base
4   default from: 'support@techcareerbooster.com'
5   layout 'mailer'
6 end
```

(the above code snippet online)

Do you see the `default` call on line 4. This is used to define default settings for the Mailers. Currently, this sets the `from` to be `support@techcareerbooster.com`. Set this to any value that you see fit.

Note also that `ApplicationMailer` specifies the default layout to be used for the emails. Just like rendering page views the views might be using a layout, views for emails are using a layout out. The default layout is the file `app/views/layouts/mailer.html.haml`, thanks to this call above `layout 'mailer'`. If you remove this line, then the default layout is the file with the same name as the Mailer name. Hence, it would have been `app/views/layouts/user_mailer.html.haml` for the `UserMailer`.

Mailer View

We have seen the implementation of the `UserMailer` (who is the equivalent to a controller). Now, let's see how we specify the view template to be used for the email content.

By default all the mailer views reside in the folder `app/views/<name_of_mailer>`. That is, we should have one view folder per custom Mailer.

Let's create the folder `app/views/user_mailer`. This will host the views of the `UserMailer` custom Mailer.

Having done that, we need to create the view for the action `ask_for_email_confirmation`. Usually, there are two files for each Mailer action. One HTML version and one TEXT version. They are both used by the mail view rendering engine to send out a multipart email, that has body both in HTML and in TEXT versions. This is recommended, because not all email clients support HTML.

Let's do that. First create the file `app/views/user_mailer/ask_for_email_confirmation.html.haml`. This is going to be the view for the action `ask_for_email_confirmation` that will be used to setup the content for the HTML part of the email. Here is the view template content:

```
- # File: app/views/user_mailer/ask_for_email_confirmation.html.haml
-
%h1
  Welcome!
%p
  = "Dear #{@user.email}"
%p
  Please, confirm your email address by clicking on this link here:
%p
  = link_to 'Click Here To Confirm Your Email Address', confirm_email_address_url\
(protocol: 'http', host: 'localhost:3000', email: @user.email, email_confirmation\
_token: @user.email_confirmation_token)
%p
  TCB Customer Success
```

(the above code snippet online)

Watch out for the following:

1. You do have access to the instance variables that you have instantiated inside the `UserMailer` action implementation. In other words, you do have access to `@user`. This is similar to the controllers instance variables that are made available to the corresponding views.
2. You have access to the named routes, like `confirm_email_address_url`. But, you will need to make sure that you use the `*_url` version and not the `*_path` version. This is because the `ActionMailer::Base` does not run in the content of HTTP request processing and does not know how to derive the whole URL. For the same reason, you have to specify the `protocol` and the `host` options.

Now, you need to create a TEXT version of your email. The TEXT version will be sent out alongside the HTML version in a multipart email body.

Let's create the file `app/views/user_mailer/ask_for_email_confirmation.text.haml`:

```
- # File: app/views/user_mailer/ask_for_email_confirmation.text.haml
-
Welcome!

= "Dear #{@user.email}"

Please, confirm your email address by clicking on this link here:

= confirm_email_address_url(protocol: 'http', host: 'localhost:3000', email: @use\
r.email, email_confirmation_token: @user.email_confirmation_token)


TCB Customer Success
```

(the above code snippet online)

As you can see above, the TEXT version is with content similar to the HTML version but without the HTML tags.

Folders And Files For Our User Mailer Action

Now, try the sign up process again. Everything should be working as expected.

Unconfirmed Email and Email Confirmation Token

Default URL Options

Remembering to add the `protocol` and `host` options on every link included in your emails is a tedious task. Also, it may be different depending on the environment. So, for example, in production, you wouldn't want these values to `http` and `localhost:3000` respectively. Rails provides a way for you to specify this per environment.

Open the file `config/environments/development.rb` and add the following line, after the other `action_mailer` related configuration:

```
...
config.action_mailer.default_url_options = {protocol: 'http', host: 'localhost:30\
00'}
...
```

(the above code snippet online)

The `config.action_mailer.default_url_options` will be used when building the URL inside the emails and when the corresponding options are not explicitly specified. And you can have different options per environment, as long as you specify this configuration differently in the corresponding environment configuration files (`config/environment/production.rb`, `config/environments/staging.rb` e.t.c.).

With the above setting in place, you can now replace the part of your User Mailer view templates that is:

```
confirm_email_address_url(protocol: 'http', host: 'localhost:3000', email: @user.\
email, email_confirmation_token: @user.email_confirmation_token)
```

(the above code snippet online)

with this:

```
confirm_email_address_url(email: @user.email, email_confirmation_token: @user.ema\
il_confirmation_token)
```

(the above code snippet online)

Make sure that you restart your Rails server before testing this feature.

If you try this feature again, you will see that the URL inside the email is built without problem. Protocol and host will be present.

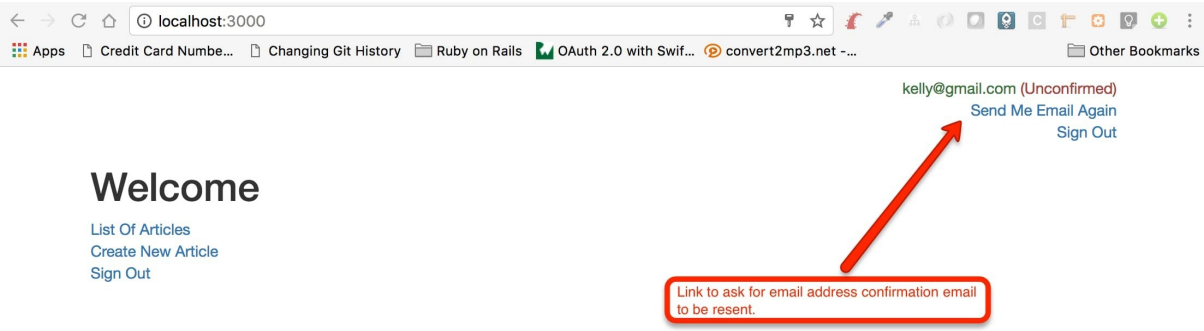
Ask For Email Address Confirmation Email To Be Resent

Now we want the user to be able to ask for the email to confirm their email address to be sent out again. Maybe, the user cannot find the original email and they want another one to be delivered to their email box.

How would we go about implementing this feature?

Action To Request Email To Be Resent

Let's add a link below the currently logged in user email, which appears at the top right. This link will be used by the user to ask for the email address confirmation email to be redelivered. Something like this:



Send Me Email Again Link

Endpoint in Routes

First, let's make sure that we have the correct endpoint defined inside the `config/routes.rb` file. Here is the version of `config/routes.rb` that includes the endpoint for this new feature:

```
# File: config/routes.rb
#
Rails.application.routes.draw do
  resources :articles

  get '/sign_up', to: 'sign_up#new'
  post '/sign_up', to: 'sign_up#create'
  get '/confirm_email_address', to: 'sign_up#confirm_email_address' # ?email=<ema\
il>&email_confirmation_token=<email_confirmation_token>
  post '/confirm_email_address_resend', to: 'sign_up#confirm_email_address_resend'

  get '/sign_in', to: 'sign_in#new'
```

```
13   post '/sign_in', to: 'sign_in#create'
14
15   get '/sign_out', to: 'sign_in#destroy'
16   delete '/sign_out', to: 'sign_in#destroy'
17
18   root 'welcome#index'
19 end
```

(the above code snippet online)

Do you see line 9? This is the new entry. You can see that we prefer here to make it a `post` request in order to make sure that we are changing things in the server state. Here is the list of routes generated:

```
1 blog $ bin/rake routes
2 ...
3 confirm_email_address_resend POST   /confirm_email_address_resend(.:format) sign_\
4 up#confirm_email_address_resend
5 ...
6 blog $
```

(the above code snippet online)

Link On Layout

Let's now create the link that would send this request. We are going to put it in the layout, inside `app/views/layouts/_signed_in_user.html.haml`:

```
1  - # File: app/views/layouts/_signed_in_user.html.haml
2  - #
3  - if current_user
4    .signed-in-user-container.text-right
5
6      .signed-in-user.text-success
7        = current_user.email
8        - unless current_user.email_confirmed?
9          %span.text-danger
10           = ' (Unconfirmed) '
11
12     - unless current_user.email_confirmed?
13       .link-to-resend-email
14         = link_to 'Send Me Email Again', confirm_email_address_resend_path, metho\
15 d: :post
16
17     .sign-out-link
18       = link_to 'Sign Out', sign_out_path, method: :delete
```

(the above code snippet online)

Look at lines 12 to 14. These have to do with the link `Send Me Email Again`. If you save this partial and visit a page after having signed in with a user that has unconfirmed email, then you will see the link present.

Action on `SignUpController`

Let's go now and implement the action `#confirm_email_address_resend` in `SignUpController`. This is how we can do that:

```
# File: app/controllers/sign_up_controller.rb
#
class SignUpController < ApplicationController
  skip_before_action :authenticate_user, except: [:confirm_email_address_resend]
  before_action :redirect_if_signed_in, except: [:confirm_email_address, :confirm\
_email_address_resend]

  def new
    @user = User.new
  end

  def create
    user = User.new(user_params)
    if user.save
      send_email_to_confirm_their_email_address(user)
      flash[:success] = 'You have successfully signed up! Please, confirm your em\
ail address. We just sent you out an email with a link to do so!'
      redirect_to root_url
    else
      @user = user
      render :new
    end
  end

  def confirm_email_address
    user = User.find_by_email_and_email_confirmation_token!(params[:email], param\
s[:email_confirmation_token])
    if user.confirm_email_address
      flash[:success] = 'You have successfully confirmed your email address!'
    else
      flash[:error] = 'Something went wrong and we cannot confirm your email addr\
ess!'
    end
    redirect_to root_url
  end

```

```
37    def confirm_email_address_resend
38      if current_user.email_confirmed?
39        flash[:warning] = 'Your email is already confirmed!'
40      else
41        send_email_to_confirm_their_email_address(user)
42        flash[:success] = 'Please, confirm your email address. We just sent you out\
43  an email with a link to do so!'
44      end
45      redirect_to root_url
46    end
47
48    private
49
50    def user_params
51      params.require(:user).permit(:email, :password, :password_confirmation)
52    end
53
54    def send_email_to_confirm_their_email_address(user)
55      UserMailer.ask_for_email_confirmation(user.id).deliver
56    end
57  end
```

(the above code snippet online)

We did changes on line 4, 5 that have to do with the before action filters. And then we introduced the implementation of the action `confirm_email_address_resend`.

1. On line 4 we had to tell that the `confirm_email_address_resend` needs the user to be authenticated.
2. On line 5 we had to tell that the same.

The implementation of the method is very simple. It relies on the `send_email_to_confirm_-their_email_address()` private method. If the currently logged in user has confirmed their email, we do nothing but displaying a warning message. Otherwise, we call `send_email_to_-confirm_their_email_address()`.

And we are done!

Here is a video that demonstrates this new feature:

Unconfirmed Email and Email Confirmation Token

Indexes In Database

We have learnt in the RDBMS section of this course that indexes are very important, because they make sure that we can search for a specific record very quickly. Also, if the index is a unique index, then it protects us from entering duplicate records in our table.

Let's see what is the status of our indexes in the current status of our blog application.

Table `articles`

If we log in to our database using `psql` we will be able to get information about the setup of the `articles` table:

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \connect blog_development
You are now connected to database "blog_development" as user "panayotismatsinopou\
los".
blog_development=# \d+ articles
                                                        Table "public.articles"
   Column   |            Type             |                      Modifiers      \
                 | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
-----------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('articles_id\
_seq'::regclass) | plain    |              |
 title      | character varying           | not null                            \
                 | extended |              |
 text       | character varying           | not null                            \
                 | extended |              |
 created_at | timestamp without time zone | not null                            \
                 | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                 | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)

blog_development=#
```

(the above code snippet online)

With the `\connect blog_development` we connect to the blog database and with the `\d+ articles` we ask information about the `articles` table.

As you can see there is a section that lists the `Indexes` that are defined on the `articles` table.

There is one index, named `articles_pkey` which is an index on the `id` column of the `articles` table. This index is a `PRIMARY KEY` index and makes sure that `id` holds unique values, i.e. is a unique index too.

Note that when we defined the migration for the `articles` table, we didn't actually specify anything about the `id` column. This was the code that we used:

```ruby
# File: db/migrate/20170519062431_create_articles.rb
#
class CreateArticles < ActiveRecord::Migration[5.1]
  def change
    create_table :articles do |t|
      t.string :title
      t.text   :text

      t.timestamps
    end
  end
end
```

(the above code snippet online)

The `id` and the fact that this is a primary key was specified automatically by the `ActiveRecord::Migration` functionality.

Table `users`

Being inside the `psql` prompt, let's now see the structure of the `users` table:

```
blog_development=#\d+ users
                                                                Table "public.use\
rs"
          Column          |            Type             |                     Mod\
ifiers                     | Storage  | Stats target | Description
--------------------------+-----------------------------+------------------------\
----------------------------+----------+--------------+-------------
 id                       | bigint                      | not null default nextva\
l('users_id_seq'::regclass) | plain    |              |
 email                    | character varying           | not null               \
                            | extended |              |
 password_digest          | character varying           | not null               \
                            | extended |              |
 created_at               | timestamp without time zone | not null               \
                            | plain    |              |
 updated_at               | timestamp without time zone | not null               \
                            | plain    |              |
 email_confirmation_token | character varying           |                        \
                            | extended |              |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)

blog_development=#
```

(the above code snippet online)

Similarly to `articles`, there is a `PRIMARY KEY` index on the `id` column of the table `users`. It was automatically specified by the `ActiveRecord::Migration`.

The point here is that table `users` is queried on the `email` column too. In other words, we have Ruby statements that try to locate a user by its `email`. For example, the `user.save` statement inside the `#create` action of the `SignUpController`. Because of the fact that we have a uniqueness validation rule on the `User` model (`validates :email, presence: true, uniqueness: { case_sensitive: false }`), before the `user.save` commits the new `user` to the database, it checks whether the user with the given `email` already exists. This check is done by querying the database on the `email` column.

Searching the table `users` by `email` will be becoming slower and slower as the number of rows in the table becomes larger. This is because of the fact that we don't have an index on this particular column.

Not only that, we better go on with creating a unique index on this column in order to make sure that we never have a table with duplicate entries.

How do we create a unique index on the `email` column of the `users` table? We have to create a new schema migration:

```
blog $ bin/rails generate migration add_index_to_users_email
      invoke  active_record
      create    db/migrate/20170617084556_add_index_to_users_email.rb
blog $
```

(the above code snippet online)

Edit the generated file and make it have the following content:

```ruby
# File: db/migrate/20170617084556_add_index_to_users_email.rb
#
class AddIndexToUsersEmail < ActiveRecord::Migration[5.1]
  def change
    add_index :users, [:email], unique: true, name: 'users_email_uidx'
  end
end
```

(the above code snippet online)

As you can read above, we use the `ActiveRecord::Migration` command `add_index`. This command takes as input the name of the table we want to add index on, the columns the index is going to index and whether it will be unique or not. Then we also give the name of the index.

Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170617084556 AddIndexToUsersEmail: migrating ============================
-- add_index(:users, [:email], {:unique=>true, :name=>"users_email_uidx"})
   -> 0.0165s
== 20170617084556 AddIndexToUsersEmail: migrated (0.0166s) ===================

blog $
```

(the above code snippet online)

Now, the index has been created and we can confirm that with the `\d+ users` command in the `psql` console:

```
blog_development=# \d+ users
                                                              Table "public.use\
rs"
          Column          |            Type             |                     Mod\
ifiers                      | Storage  | Stats target | Description
--------------------------+-----------------------------+------------------------\
----------------------------+----------+--------------+-------------
 id                       | bigint                      | not null default nextva\
l('users_id_seq'::regclass) | plain    |              |
 email                    | character varying           | not null               \
                            | extended |              |
 password_digest          | character varying           | not null               \
                            | extended |              |
 created_at               | timestamp without time zone | not null               \
                            | plain    |              |
 updated_at               | timestamp without time zone | not null               \
                            | plain    |              |
 email_confirmation_token | character varying           |                        \
                            | extended |              |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_email_uidx" UNIQUE, btree (email)

blog_development=#
```

(the above code snippet online)

Do you see the new line that has been added in the indexes section? The `"users_email_uidx" UNIQUE, btree (email)` confirms that we have added the index on the `email` column of the table. And this index is a unique index.

Experimenting with Uniqueness

Before we leave this section, and while being in the `psql` section, let's try to create a new user with same email as an existing one but with different case.

Currently, the users that I have in my database are the following:

```
blog_development=# select id, email from users order by email;
 id |       email
----+-------------------
 18 | angela@gmail.com
  9 | bar@gmail.com
  8 | foo@gmail.com
 20 | john2@gmail.com
 12 | john@gmail.com
 21 | kelly@gmail.com
 17 | kristin@gmail.com
 10 | mary@gmail.com
 16 | paul@gmail.com
 19 | peter2@gmail.com
 13 | peter@gmail.com
(11 rows)

blog_development=#
```

(the above code snippet online)

Using standard sql, I will try to create the user "PETER@gmail.com":

```
blog_development=# insert into users (email, password_digest, created_at, updated\
_at) values ('PETER@gmail.com', 'password-hash', current_timestamp, current_times\
tamp);
INSERT 0 1
blog_development=# select id, email from users order by id;
 id |       email
----+-------------------
  8 | foo@gmail.com
  9 | bar@gmail.com
 10 | mary@gmail.com
 12 | john@gmail.com
 13 | peter@gmail.com
 16 | paul@gmail.com
 17 | kristin@gmail.com
 18 | angela@gmail.com
 19 | peter2@gmail.com
 20 | john2@gmail.com
 21 | kelly@gmail.com
 22 | PETER@gmail.com
(12 rows)

blog_development=#
```

(the above code snippet online)

Oooops! What happened? `PETER@gmail.com` user was created even if `peter@gmail.com` was already in. This is standard behaviour in PostgreSQL. The sorting and comparison of strings is case sensitive. This might be a problem, if you don't want strings to be compared in a case sensitive way. But only if you do db-level operations using standard SQL.

Let's delete this newly created user:

```
blog_development=# delete from users where email = 'PETER@gmail.com';
DELETE 1
blog_development=#
```

(the above code snippet online)

And let's try to add another one with email `peter@gmail.com`:

```
blog_development=# insert into users (email, password_digest, created_at, updated\
_at) values ('peter@gmail.com', 'password-hash', current_timestamp, current_times\
tamp);
ERROR:  duplicate key value violates unique constraint "users_email_uidx"
DETAIL:  Key (email)=(peter@gmail.com) already exists.
blog_development=#
```

(the above code snippet online)

In this case, the unique index protected us from adding the same email twice.

Now exit the `psql` console and start a rails console. Then try to create a user with email `"PETER@gmail.com"`.

```
blog_development=# \q
blog $ bin/rails console
Loading development environment (Rails 5.1.1)
2.3.3 :001 > User.create!(email: 'PETER@gmail.com', password: '12345678')
   (0.2ms)  BEGIN
  User Exists (0.8ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "PETER@gmail.com"], ["LIMIT", 1]]
   (0.2ms)  ROLLBACK
ActiveRecord::RecordInvalid: Validation failed: Email has already been taken
        from (irb):1
2.3.3 :002 >
```

(the above code snippet online)

As you can see, the problem that we encountered with inserting the user `PETER@gmail.com` via standard SQL didn't occur when we tried to create the same user using Ruby `ActiveRecord` statement. This is because of the protection that we have with the uniqueness validation on the model.

Having done the above experiment, then it is obvious that you should try to avoid creating data into your database using standard SQL, but instead try to use the Ruby statements. Ruby statements take advantage of the validation rules that we have designed at the Ruby application layer, whereas standard SQL statements do not.

> **Important**: The behaviour that we have encountered with PostgreSQL is not the same for all other database management systems. For example, MySQL has case insensitive string comparison and sorting, by default.

SMTP Server For Production Environment

Now that we have finished with the email confirmation feature, we need to commit our code and push to our remote Github account. Also, we need to push it to our Heroku remote so that the application is deployed to production..

The point is that in order for us to have this new feature in production, we first need to integrate a real SMTP server.

Since your application is deployed in Heroku, one good solution is to use the add-on for integration with SendGrid.

Visit your Heroku dashboard and your blog application `Resources` tab. There, make sure that you add the free version of the SendGrid add-on.

After you do that, then you will have to edit the file `config/environments/production.rb` to have the following entries that are related to `ActionMailer`:

```
  config.action_mailer.default_url_options   = {protocol: 'https', host: 'panosm-\
blog.herokuapp.com'}
  config.action_mailer.raise_delivery_errors = false
  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = {
      address:              'smtp.sendgrid.com',
      port:                 587,
      domain:               'heroku.com',
      user_name:            ENV['SENDGRID_USERNAME'],
      password:             ENV['SENDGRID_PASSWORD'],
      authentication:       'plain',
      openssl_verify_mode:  'none',
      enable_starttls_auto: true  }
```

(the above code snippet online)

Note the first line needs to have the `host` for **your** application. What you see above is the `host` value for my application.

Commit your code to git and then push to `origin master`. Also push to `heroku master` in order to deploy your changes to your production application too.

The following commands are necessary in order to push your code to production (assuming that you have commit your changes to your git repository):

```
blog $ git push heroku master
....
blog $ heroku run rake db:migrate --app panos-blog
...
blog $ heroku ps:restart --app panos-blog
...
blog $
```

(the above code snippet online)

Then try to sign up and confirm your email address. You will see that you will receive an email to be asking for email confirmation.

Closing Note

We have seen what are the tools that Rails offers to us in order to be able to send emails out.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

Task A

All the email integration described in the chapter for your `blog` application needs to be pushed to your remote Github repository and deployed to your production Heroku application.

Task B

Continue working on the e-shop that you have started since the beginning of this section. You need to make sure that you offer the same email-related features that we have developed for `blog` in this chapter:

1. Users should confirm their email address.
2. Users should be able to ask for the email to confirm their email address to be resent.

Push your code to your remote Github repository and to your production Heroku e-shop application.

5 - Authorization

Summary

Rails Authorization

In this chapter, we proceed with the other aspect of security, the one that is called *Authorization*. Authorized users are users that have access to the Web application but they are restricted to specific actions and areas of it, based on their privileges. You will learn how to implement authorization in a Rails application protecting specific actions and pages from unauthorized users.

Learning Goals

1. Learn how to bind an Article to a User, i.e. create foreign keys and relationships between objects.
2. Learn how to add a new column to a table.
3. Learn how to add a new foreign key constraint.
4. Learn how to write schema migrations that can be rolled back.
5. Learn how to explicitly tell which statements should be used for migration and which for rollback.

6. Learn how to create an index on a column of a table.
7. Learn how you can specify the foreign key constraint name.
8. Learn how you can specify the index name.
9. Learn how to create a column that is foreign key and it has an index on it, using a single command.
10. Learn how to tell Rails how a part of migration can be rolled back.
11. Learn how to associate models using Ruby statements.
12. Learn how to update your controller to create the associations between objects.
13. Learn how to update your controller to allow actions on specific Users, i.e. to apply authorization.
14. Learn how to hide and show links based on whether the User is authorized or not.

Introduction

Currently, our users can sign up and sign in. Sign in means that the Web application knows who is the *current user*.

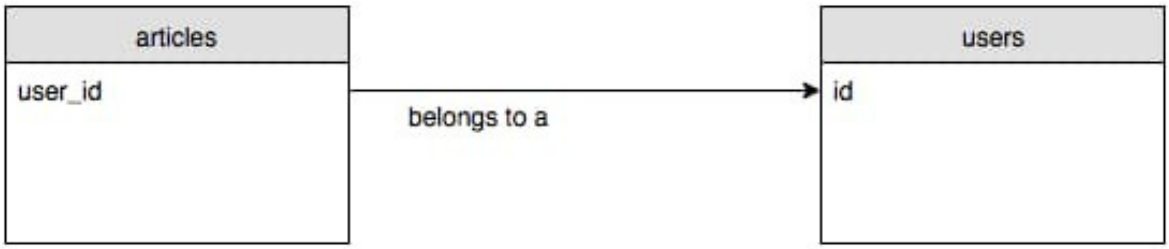
All of our users have the same privileges, which might be a problem. For example, currently, the user that has created the Article "Ruby Hashes", for example, can see their Article being deleted by another user that might sign in. In other words, a user has CUD (Create, Update, Delete) access to other user's Articles.

What we really need here is to make sure that an Article can only be CUDed only by the user that has created the Article. In other words, what we want is what is generally called *Authorization*. You may have access to a Web application, but you may not have the right to do anything. Only specific actions on specific resources.

Binding Articles To Users

In order to protect an Article from being edited by a User that has not created the particular Article, we will need to make sure that we know who is the actual User that has created the particular Article.

We need something like this:



Binding Articles To Users

In other words, we need a column in `articles` table that is going to be pointing to the owning `users` table row. It's going to be a foreign key and we will call it `user_id`.

Important: The existing articles in our database is a *problem* because we don't know who they belong to. So, we will not be able to map them to a User. For that reason, we will delete them. This is not an issue for us, since our blog application is still under development.

Schema Migration

In order to change the table `articles` we will need to write a new schema migration:

```
blog $ bin/rails generate migration add_user_id_to_articles
      invoke  active_record
      create    db/migrate/20170618060653_add_user_id_to_articles.rb
blog $
```

(the above code snippet online)

Then let's edit the schema migration file to be as follows:

```
1  # File: db/migrate/20170618060653_add_user_id_to_articles.rb
2  #
3  class AddUserIdToArticles < ActiveRecord::Migration[5.1]
4    def change
5      # delete all Articles first
6      say_with_time('Deleting all articles') do
7        Article.destroy_all
8      end
9
10     add_column :articles, :user_id, :bigint, null: false
11     add_foreign_key(:articles, :users)
12   end
13 end
```

(the above code snippet online)

Lines 6 to 8, they are there to delete the existing Articles. The `say_with_time` statement allows us to execute an `ActiveRecord::Migration`, the one given inside the block to this call, and displaying a descriptive informative message at the same time. Also, it calculates and prints the time that it is needed to execute the block of code.

The main migration statements are the ones in lines 10 and 11. The `add_column` adds the `user_id` column and the `add_foreign_key` turns the `user_id` to a foreign key.

Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170618060653 AddUserIdToArticles: migrating ==============================
-- Deleting all articles
   -> 0.0157s
-- add_column(:articles, :user_id, :bigint, {:null=>false})
   -> 0.0021s
-- add_foreign_key(:articles, :users)
   -> 0.0031s
== 20170618060653 AddUserIdToArticles: migrated (0.0210s) =====================

blog $
```

(the above code snippet online)

Migration ran successfully! If you start a `psql` session and inspect the structure of the `articles` table, you will see this:

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \connect blog_development
You are now connected to database "blog_development" as user "panayotismatsinopou\
los".
blog_development=# \d+ articles
                                                        Table "public.articles"
   Column   |            Type             |                      Modifiers      \
                 | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
-----------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('articles_id\
_seq'::regclass) | plain    |              |
 title      | character varying           | not null                            \
                 | extended |              |
 text       | character varying           | not null                            \
                 | extended |              |
 created_at | timestamp without time zone | not null                            \
                 | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                 | plain    |              |
 user_id    | bigint                      | not null                            \
                 | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_rails_3d31dad1cc" FOREIGN KEY (user_id) REFERENCES users(id)
```

```
30
31 blog_development=#
```

(the above code snippet online)

As you can see, the column `user_id` has been added and a new `Foreign-key constraint` has been created.

However, I can see two problems here. One minor and one major.

Minor problem: The name of the foreign key constraint is `fk_rails_3d31dad1cc` which is not very descriptive. If this constraint fails and we see an error raised by this, we will not be able to quickly identify which columns are involved in this constraint. We would have liked something more like the following: `articles_user_id_to_users_id_fk`.

Major problem: There isn't index on the `user_id` column. Hence, when we will be looking for articles of a particular user, then query will be slow doing full scan on the table `articles` to locate the correct rows.

> **Important**: Some RDBMSes create an index for each foreign key that you create. This is what MySQL does for example.

How can we fix these two issues? We have two options:

1. Create a new schema migration that would rename the foreign key and also add the index OR
2. To *rollback* this schema migration, correct the content of the migration to include the correct name for foreign key and the index creation and then run the migration again.

Rolling Back Latest Schema Migration

We will go with the 2nd option for tutorial reasons. We want to show you how you can rollback the latest schema migration.

For a schema migration to be rolled back, we need to let `ActiveRecord::Migration` know what are the actions that need to take place in order for the things that migration has done to be undone.

Usually, `ActiveRecord::Migration` can tell it automatically from what is inside the `change` method. So, if inside the `change` method we have an `add_column` statement to create a column, `ActiveRecord::Migration` knows that rolling back can be achieved with a call to `remove_column`. Or the `add_foreign_key` can be rolled back with a corresponding `remove_foreign_key` command.

Our `change` implementation for the addition of `user_id` to `Articles`, includes

1. the deletion of the existing articles.
2. the `add_column` statement to add the new column.
3. the `add_foreign_key` statement to add the foreign key constraint.

`ActiveRecord::Migration` can automatically derive the roll back actions for all these things except for the first one. It cannot know how to undo the deletion of the articles.

Let's put this statement of the `change` command into comments for the time being:

```ruby
# File: db/migrate/20170618060653_add_user_id_to_articles.rb
#
class AddUserIdToArticles < ActiveRecord::Migration[5.1]
  def change
    # delete all Articles first
    # say_with_time('Deleting all articles') do
    #   Article.destroy_all
    # end

    add_column :articles, :user_id, :bigint, null: false
    add_foreign_key(:articles, :users)
  end
end
```

(the above code snippet online)

Now the `change` method contains actions that `ActiveRecord::Migration` knows how to roll back. Having said that, we can ask a roll back:

```
blog $ bin/rake db:rollback
== 20170618060653 AddUserIdToArticles: reverting ==============================
-- remove_foreign_key(:articles, :users)
   -> 0.0126s
-- remove_column(:articles, :user_id, :bigint, {:null=>false})
   -> 0.0011s
== 20170618060653 AddUserIdToArticles: reverted (0.0198s) =====================

blog $
```

(the above code snippet online)

The rollback takes place by `ActiveRecord::Migration`, by applying the reverse action of the actions included in the `change` method. Also, note that these reverse actions are executed in the opposite order that they were defined inside the `change` method. First the removal of the foreign key, and then the removal of the column.

If you don't want to rely on `ActiveRecord::Migration` to derive the roll back actions from the `change` statements, you can, instead, explicitly tell `ActiveRecord::Migration` which statements should be used for `migrate` and which statements should be used for `rollback`.

Here it is how:

```ruby
# File: db/migrate/20170618060653_add_user_id_to_articles.rb
#
class AddUserIdToArticles < ActiveRecord::Migration[5.1]
  def up
    # delete all Articles first
    say_with_time('Deleting all articles') do
      Article.destroy_all
    end

    add_column :articles, :user_id, :bigint, null: false
    add_foreign_key(:articles, :users)
  end

  def down
    remove_foreign_key(:articles, :users)
    remove_column :articles, :user_id
  end
end
```

(the above code snippet online)

Instead of having a `change` method, we have an `up` method that corresponds to the `migrate` actions and a `down` method which corresponds to the `rollback` actions.

Having done that, we can now tell `ActiveRecord::Migration` that, when we `rollback`, we don't care about reintroducing the Articles deleted on the `migrate` direction.

With the above version of our `AddUserIdToArticles` migration, let's run the migration again:

```
blog $ bin/rake db:migrate
== 20170618060653 AddUserIdToArticles: migrating ==============================
-- Deleting all articles
   -> 0.0066s
-- add_column(:articles, :user_id, :bigint, {:null=>false})
   -> 0.0023s
-- add_foreign_key(:articles, :users)
   -> 0.0037s
== 20170618060653 AddUserIdToArticles: migrated (0.0128s) =====================

blog $
```

(the above code snippet online)

And then let's roll it back:

```
blog $ bin/rake db:rollback
== 20170618060653 AddUserIdToArticles: reverting ==============================
-- remove_foreign_key(:articles, :users)
   -> 0.0060s
-- remove_column(:articles, :user_id)
   -> 0.0003s
== 20170618060653 AddUserIdToArticles: reverted (0.0065s) =====================

blog $
```

(the above code snippet online)

You can clearly see which actions are being executed in the forward/up direction and which actions are being executed in the backward/down direction.

Foreign Key Name and Index

Let's now change the content of the file with the schema migration so that it creates the foreign key with the name that we want and that it also creates an index on the `user_id` on the `articles` table:

```
1  # File: db/migrate/20170618060653_add_user_id_to_articles.rb
2  #
3  class AddUserIdToArticles < ActiveRecord::Migration[5.1]
4    def up
5      # delete all Articles first
6      say_with_time('Deleting all articles') do
7        Article.destroy_all
8      end
9
10     add_column :articles, :user_id, :bigint, null: false
11     add_foreign_key(:articles, :users, name: 'articles_user_id_users_id_fk')
12     add_index :articles, [:user_id], name: 'articles_user_id_idx'
13   end
14
15   def down
16     remove_index :articles, name: 'articles_user_id_idx'
17     remove_foreign_key(:articles, name: 'articles_user_id_users_id_fk')
18     remove_column :articles, :user_id
19   end
20 end
```

(the above code snippet online)

You can see lines 11 and 12. We specify the name of the foreign key constraint and we have a statement to add the index. See also how we have adapted the rollback/down statements accordingly.

Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170618060653 AddUserIdToArticles: migrating ==============================
-- Deleting all articles
   -> 0.0088s
-- add_column(:articles, :user_id, :bigint, {:null=>false})
   -> 0.0026s
-- add_foreign_key(:articles, :users, {:name=>"articles_user_id_users_id_fk"})
   -> 0.0042s
-- add_index(:articles, [:user_id], {:name=>"articles_user_id_idx"})
   -> 0.0036s
== 20170618060653 AddUserIdToArticles: migrated (0.0194s) =====================

blog $
```

(the above code snippet online)

The migrations ran successfully. Let's inspect the structure of the table `articles` using `psql`:

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \connect blog_development
You are now connected to database "blog_development" as user "panayotismatsinopou\
los".
blog_development=# \d+ articles
                                                          Table "public.articles"
   Column   |            Type             |                      Modifiers      \
                 | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
-----------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('articles_id\
_seq'::regclass) | plain    |              |
 title      | character varying           | not null                            \
                 | extended |              |
 text       | character varying           | not null                            \
                 | extended |              |
 created_at | timestamp without time zone | not null                            \
                 | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                 | plain    |              |
 user_id    | bigint                      | not null                            \
                 | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)
    "articles_user_id_idx" btree (user_id)
Foreign-key constraints:
```

```
    "articles_user_id_users_id_fk" FOREIGN KEY (user_id) REFERENCES users(id)

blog_development=#
```

(the above code snippet online)

You can see now the name of the foreign key constraint, as well as the index that has been created on the column `user_id`.

Add Reference Does It All

Let's now rollback again:

```
blog $ bin/rake db:rollback
== 20170618060653 AddUserIdToArticles: reverting ==============================
-- remove_index(:articles, {:name=>"articles_user_id_idx"})
   -> 0.0012s
-- remove_foreign_key(:articles, {:name=>"articles_user_id_users_id_fk"})
   -> 0.0068s
-- remove_column(:articles, :user_id)
   -> 0.0005s
== 20170618060653 AddUserIdToArticles: reverted (0.0086s) =====================

blog $
```

(the above code snippet online)

Now that we have understood what it takes to create a proper reference from `articles` table to `users` table, with the three schema migration commands, we will tell you that Rails `ActiveRecord::Migration` allows you to do that with one command, i.e. one command to carry out all the three commands at once.

Change your schema migration file as follows:

```
# File: db/migrate/20170618060653_add_user_id_to_articles.rb
#
class AddUserIdToArticles < ActiveRecord::Migration[5.1]
  def up
    # delete all Articles first
    say_with_time('Deleting all articles') do
      Article.destroy_all
    end

    add_reference :articles, :user, foreign_key: {name: 'articles_user_id_users_i\
d_fk'}, index: {name: 'articles_user_id_idx'}
  end

```

```
  def down
    remove_reference :articles, :user, foreign_key: {name: 'articles_user_id_user\
s_id_fk'}, index: {name: 'articles_user_id_idx'}
  end
end
```

(the above code snippet online)

We now use the `add_reference` command that allows you to specify the foreign key name and the index to be created. And on the `down` version we just call `remove_reference` instead.

Run the schema migration:

```
blog $ bin/rake db:migrate
== 20170618060653 AddUserIdToArticles: migrating ==============================
-- Deleting all articles
   -> 0.0130s
-- add_reference(:articles, :user, {:foreign_key=>{:name=>"articles_user_id_users\
_id_fk"}, :index=>{:name=>"articles_user_id_idx"}})
   -> 0.0183s
== 20170618060653 AddUserIdToArticles: migrated (0.0315s) =====================

blog $
```

(the above code snippet online)

And then inspect the structure of the table `articles` using `psql`:

```
                                                        Table "public.articles"
   Column   |            Type             |                      Modifiers      \
                 | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
-----------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('articles_id\
_seq'::regclass) | plain    |              |
 title      | character varying           | not null                            \
                 | extended |              |
 text       | character varying           | not null                            \
                 | extended |              |
 created_at | timestamp without time zone | not null                            \
                 | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                 | plain    |              |
 user_id    | bigint                      |                                     \
                 | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)
    "articles_user_id_idx" btree (user_id)
```

```
Foreign-key constraints:
    "articles_user_id_users_id_fk" FOREIGN KEY (user_id) REFERENCES users(id)
```

(the above code snippet online)

You can see that the `user_id` has been created. Also, the `articles_user_id_idx` index on column `user_id` and the foreign key constraint with the name that we like.

Pretty cool! Isn't it?

Avoid Repetition in Migration

Finally, we can see that line 10 is similar to line 14, except from the fact that we call different `ActiveRecord::Migration` statement. This statement, the `add_reference` is reversible and `ActiveRecord::Migration` knows how to roll it back. Let's take advantage of it.

First, roll back again.

```
blog $ bin/rake db:rollback
== 20170618060653 AddUserIdToArticles: reverting ==============================
-- remove_reference(:articles, :user, {:foreign_key=>{:name=>"articles_user_id_us\
ers_id_fk"}, :index=>{:name=>"articles_user_id_idx"}})
   -> 0.0189s
== 20170618060653 AddUserIdToArticles: reverted (0.0409s) =====================

blog $
```

(the above code snippet online)

Also, inspect with `psql`. You will see that everything has been rolled back as expected:

```
blog_development=# \d+ articles
                                                        Table "public.articles"
   Column   |            Type             |                      Modifiers      \
                 | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
-----------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('articles_id\
_seq'::regclass) | plain    |              |
 title      | character varying           | not null                            \
                 | extended |              |
 text       | character varying           | not null                            \
                 | extended |              |
 created_at | timestamp without time zone | not null                            \
                 | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                 | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)
```

(the above code snippet online)

Now, update the content of the migration file as follows:

```
# File: db/migrate/20170618060653_add_user_id_to_articles.rb
#
class AddUserIdToArticles < ActiveRecord::Migration[5.1]
  def change
    reversible do |dir|
      dir.up do
        # delete all Articles first
        say_with_time('Deleting all articles') do
          Article.destroy_all
        end
      end
    end

    add_reference :articles, :user, foreign_key: {name: 'articles_user_id_users_i\
d_fk'}, index: {name: 'articles_user_id_idx'}
  end
end
```

(the above code snippet online)

Now we only use `change` instead of both `up` and `down`. And we put the deletion of Articles inside a `dir.up` block which is also inside a `reversible do` block. The idea here is that whatever is not possible for `ActiveRecord::Migration` to derive a roll back statement for, we put it in `reversible do` and we tell explicitly which statements will be executed in the `up` direction and which will be executed in the `down` direction. In our example, we said that the deletion of the Articles will take place only in the `up` direction. We didn't give anything for the `down` direction (there is no `dir.down` block). On the other hand, we leave outside the `reversible` block whatever `ActiveRecord::Migration` can derive a roll back statement for.

Now do the migration:

```
blog $ bin/rake db:migrate
== 20170618060653 AddUserIdToArticles: migrating ==============================
-- Deleting all articles
   -> 0.0063s
-- add_reference(:articles, :user, {:foreign_key=>{:name=>"articles_user_id_users\
_id_fk"}, :index=>{:name=>"articles_user_id_idx"}})
   -> 0.0089s
== 20170618060653 AddUserIdToArticles: migrated (0.0153s) =====================

blog $
```

(the above code snippet online)

And then do the rollback:

```
blog $ bin/rake db:rollback
== 20170618060653 AddUserIdToArticles: reverting ==============================
-- remove_reference(:articles, :user, {:foreign_key=>{:name=>"articles_user_id_us\
ers_id_fk"}, :index=>{:name=>"articles_user_id_idx"}})
   -> 0.0067s
== 20170618060653 AddUserIdToArticles: reverted (0.0108s) =====================

blog $
```

(the above code snippet online)

This confirms that we have written our migration correctly.

Before we leave this paragraph, let's run the migration again so that we have the `user_id` in `articles` table:

```
blog $ bin/rake db:migrate
== 20170618060653 AddUserIdToArticles: migrating ==============================
-- Deleting all articles
   -> 0.0171s
-- add_reference(:articles, :user, {:foreign_key=>{:name=>"articles_user_id_users\
_id_fk"}, :index=>{:name=>"articles_user_id_idx"}})
   -> 0.0238s
== 20170618060653 AddUserIdToArticles: migrated (0.0413s) =====================

blog $
```

(the above code snippet online)

Updating `Article` and `User` Models

We have prepared our database to store, inside `articles` table, the reference to the `users` row that creates an `Article`. But this is not enough. We need to enhance our `Article` and `User` models to have some application-level rules.

`Article`

Let's start from the `Article` model. This is the new version of it:

```
1 # File: app/models/article.rb
2 #
3 class Article < ApplicationRecord
4   belongs_to :user
5
6   validates :user, presence: true
7   validates :title, presence: true
8   validates :text, presence: true
9 end
```

(the above code snippet online)

We have added line 4. This is to tell Rails that an `Article` belongs to a `User`. We use the `belongs_to` on the model that owns the foreign key. Now that we have added this declaration here, we can write code like this:

```
article = Article.first
puts "Article #{article.id} has been created by user #{article.user.email}"
```

(the above code snippet online)

The `article.user` is a method that is defined behind the scenes thanks to the `belongs_to :user` declaration. It retrieves the User instance that corresponds to the `user_id` stored in the particular `Article`.

On line 6, we have added the validation that makes sure that an Article belongs to a user. In other words, we will not be able to create a valid `Article` unless we specify the `User` instance the article belongs to.

User

But, we may also want to easily find the Articles of a User. Wouldn't be useful if we were able to do something like this?

```
user = User.first
user.articles
```

(the above code snippet online)

The `user.article` would bring the Articles that belong to the particular User that we have in the variable `user`.

Let's enhance our `User` model in order to support this:

```
1  # File app/models/user.rb
2  #
3  class User < ApplicationRecord
4    has_many :articles
5
6    has_secure_password
7
8    validates :email, presence: true, uniqueness: { case_sensitive: false }
9
10   before_create :generate_email_confirmation_token
11
12   def email_confirmed?
13     !email_confirmation_token.present?
14   end
15
16   def confirm_email_address
17     self.email_confirmation_token = nil
18     save
19   end
20
21   def can_write_article?(article)
22     article.user.id == id
23   end
24
25   private
26
27   def generate_email_confirmation_token
28     self.email_confirmation_token = SecureRandom.urlsafe_base64
29   end
30 end
```

(the above code snippet online)

The amendment here is only the line 4, `has_many :articles`. This gives you the ability to fetch the Articles of a User when you have the User reference. Like in the following Rails console session:

```
1  2.3.3 :001 > user = User.find_by_email('bar@gmail.com')
2    User Load (0.5ms)  SELECT  "users".* FROM "users" WHERE "users"."email" = $1 LI\
3  MIT $2  [["email", "bar@gmail.com"], ["LIMIT", 1]]
4   => #<User id: 9, email: "bar@gmail.com", password_digest: "$2a$10$g/4AILw8UXB1s8\
5  G01M8P1efgzs0iZfNfw.BxdHW2HNI...", created_at: "2017-06-04 13:51:46", updated_at:\
6   "2017-06-11 14:49:49", email_confirmation_token: nil>
7  2.3.3 :002 > user.articles.each do |article|
8  2.3.3 :003 >     puts article.title
9  2.3.3 :004?>   end
10   Article Load (0.4ms)  SELECT "articles".* FROM "articles" WHERE "articles"."use\
```

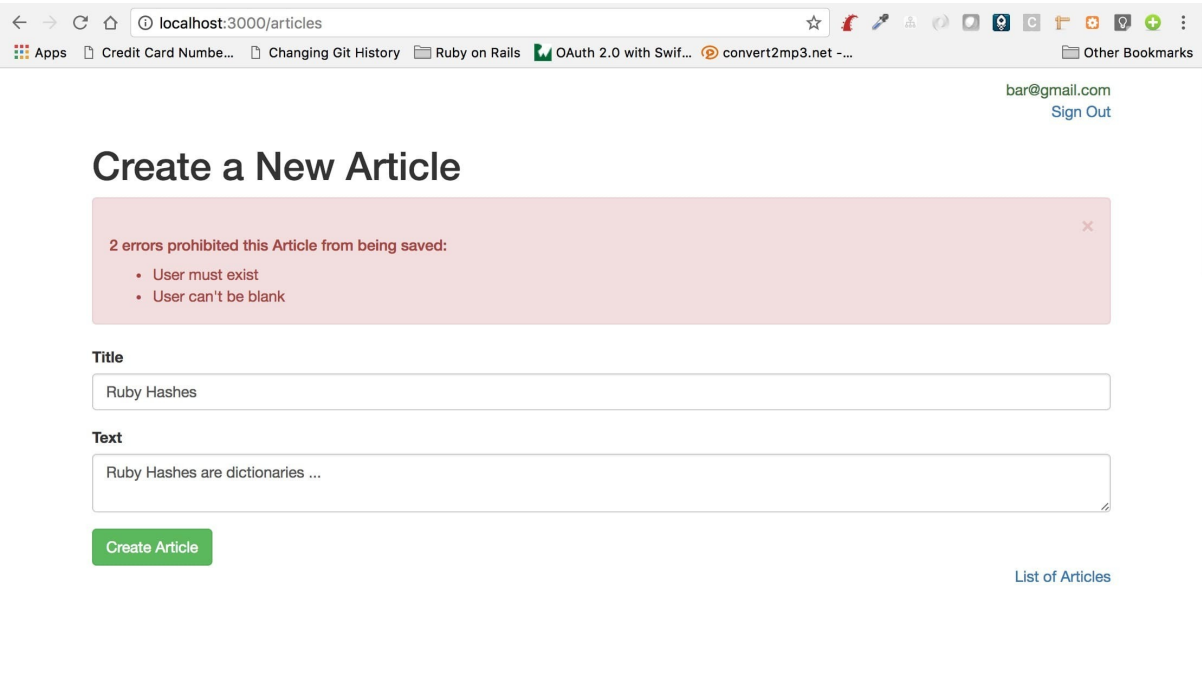
```
r_id" = $1  [["user_id", 9]]
Ruby Hashes-wrong
 => [#<Article id: 29, title: "Ruby Hashes-wrong", text: "Ruby Hashes are diction\
aries-wrong ...", created_at: "2017-06-18 10:01:27", updated_at: "2017-06-18 10:1\
6:07", user_id: 9>]
2.3.3 :005 >
```

(the above code snippet online)

As you can see, in the command `002`, we start an iteration of the Articles of the User stored in `user`. We do that with the help of the `articles` method, which is created, behind the scenes, thanks to the `has_many :articles` statement in the `User` model.

Updating the Controller

With the current status of our code, make sure you restart your Rails server and try to create a new `Article`. You will get an error like this:



Error While Creating an Article Without User

These errors are expected because the form that creates an Article does not provide for the User to bind the Article to. Actually, the problem is not on the form. We will not allow the person that creates the Article to select the User of the Article. The User of the Article being created should be put automatically to be the currently longed in User. Hence, the change needs to take place at the `ArticlesController` side. The action `ArticlesController#create` should create an `Article` bound to the currently signed in user. This is the new version of the `ArticlesController#create` action:

```
...
  def create
    article = Article.new(article_params.merge({user: current_user}))
    if article.save
      flash[:success] = 'Article has been created!'
      redirect_to edit_article_url(article)
    else
      @article = article
      render :new
    end
  end
...
```

(the above code snippet online)

The only difference is on line that instantiates the `Article`. We merge the `article_params` `Hash` with the `Hash` `{user: current_user}`. Hence, the currently signed in user is set as the user for the newly created `Article`.

If you try to create an article now, it will be created without problem and the `Article` will be bound to the currently signed in user.

Displaying Owner Next To Each Article Entry

Before we continue with further authorization cases, we will first amend the `app/views/articles/index.html.ha` file so that it displays the owner of the Articles listed.

Here is the new version of the `app/views/articles/index.html.haml` file:

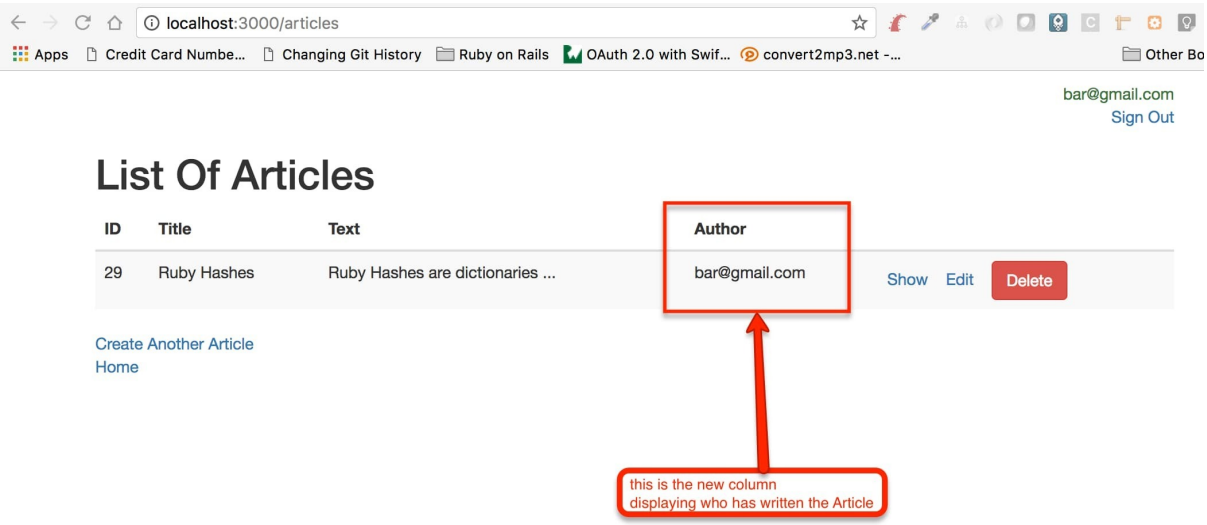
```
- # File: app/views/articles/index.html.haml
- #
%h1 List Of Articles
%table.table.table-striped
  %thead
    %tr
      %th ID
      %th Title
      %th Text
      %th Author
      %th
  %tbody
    - @articles.each do |article|
      %tr
        %td= article.id
        %td= article.title
        %td= article.text
        %td= article.user.email
```

```
19          %td
20            = link_to 'Show', article_path(article)
21              
22            = link_to 'Edit', edit_article_path(article) if current_user
23              
24            = link_to 'Delete', article_path(article), method: :delete, class: 'btn\
25  btn-danger', data: {confirm: 'Are you sure that you want to delete this Article?\
26  '} if current_user
27  - if current_user
28    %div
29      = link_to 'Create Another Article', new_article_path
30  %div
31    = link_to 'Home', root_path
```

(the above code snippet online)

On line 10 we have added the header of the new column, this being `Author`. And on line 18, we have added the value of the email of the author of the article.

Here is an example output of the page List of Articles:



Displaying Author Name In List of Articles

Editing/Updating an Article

Currently, the update method allows a User X to update the `Article` created by the `User` Y. This is because the `update` action on `ArticlesController` does not take into account who is the currently signed in User vs the owner of the `Article` being updated.

The following video demonstrates the problem.

User Peter Updates Article of User Bar

The problem that you see above applies to all three actions of `ArticlesController` that have to do with CUD on `Article` resource:

1. `edit`
2. `update`
3. `destroy`

For these actions we will hook a before action filter that would ask the User to be authorized. Here is the new version of our `ArticlesController`:

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
  skip_before_action :authenticate_user, only: [:index, :show]
  before_action :find_article, only: [:show, :edit, :update, :destroy]
  before_action :authorize_user, except: [:new, :create, :index, :show]

  def new
    @article = Article.new
  end

  def create
    article = Article.new(article_params.merge({user: current_user}))
    if article.save
      flash[:success] = 'Article has been created!'
      redirect_to edit_article_url(article)
    else
      @article = article
      render :new
    end
  end

  def index
    @articles = Article.all
  end

  def update
    if @article.update(article_params)
      flash[:success] = 'Article has been updated!'
      redirect_to edit_article_path(@article)
    else
      render :edit
    end
  end

  def destroy
```

```
    @article.destroy
    flash[:success] = 'Article has been destroyed!'
    redirect_to articles_path
  end

  private

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

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

  def authorize_user
    unless @article.user_id == current_user.id
      flash[:warning] = 'Your are not allowed to act on this Article!'
      redirect_to root_url
    end
  end
end
```

(the above code snippet online)

We have added the private method `authorize_user` and we have also added the `before_action` filter that calls this authorization method for all actions except the `:new`, `:create`, `:index` and `:show`. The `authorize_user` implementation is very simple. As long as the `@article` being updated or deleted has owner/author being equal to the currently logged in user, everything proceeds as before. Otherwise, we break the request processing loop by redirecting back to root page of the application with the appropriate message flashing as warning.

With the above implementation at hand, when user `Peter` tries to edit the article of the user `Bar`, he will get the warning and will not be able to access the edit page.

User Peter Updates Article of User Bar

Hide Edit Link For Other User's Articles

We have managed to protect the actions from being called by unauthorized users. Then, what we can do on top of that is to hide the links that are not applicable based on the authorizations the current user has.

Hide Links In List Of Articles

In the list of Articles, we have the `Edit` link and the `Delete` button that need to be hidden if the currently signed in user does not own the corresponding Article. Let's change the `app/views/articles/index.html.haml` accordingly:

```
1  - # File: app/views/articles/index.html.haml
2  - #
3  %h1 List Of Articles
4  %table.table.table-striped
5    %thead
6      %tr
7        %th ID
8        %th Title
9        %th Text
10       %th Author
11       %th
12   %tbody
13     - @articles.each do |article|
14       %tr
15         %td= article.id
16         %td= article.title
17         %td= article.text
18         %td= article.user.email
19         %td
20           = link_to 'Show', article_path(article)
21             
22           = link_to 'Edit', edit_article_path(article) if current_user && current\
23 _user.can_edit_article?(article)
24             
25           = link_to 'Delete', article_path(article), method: :delete, class: 'btn\
26  btn-danger', data: {confirm: 'Are you sure that you want to delete this Article?\
27 '} if current_user && current_user.can_delete_article?(article)
28 - if current_user
29   %div
30     = link_to 'Create Another Article', new_article_path
31 %div
32   = link_to 'Home', root_path
```

(the above code snippet online)

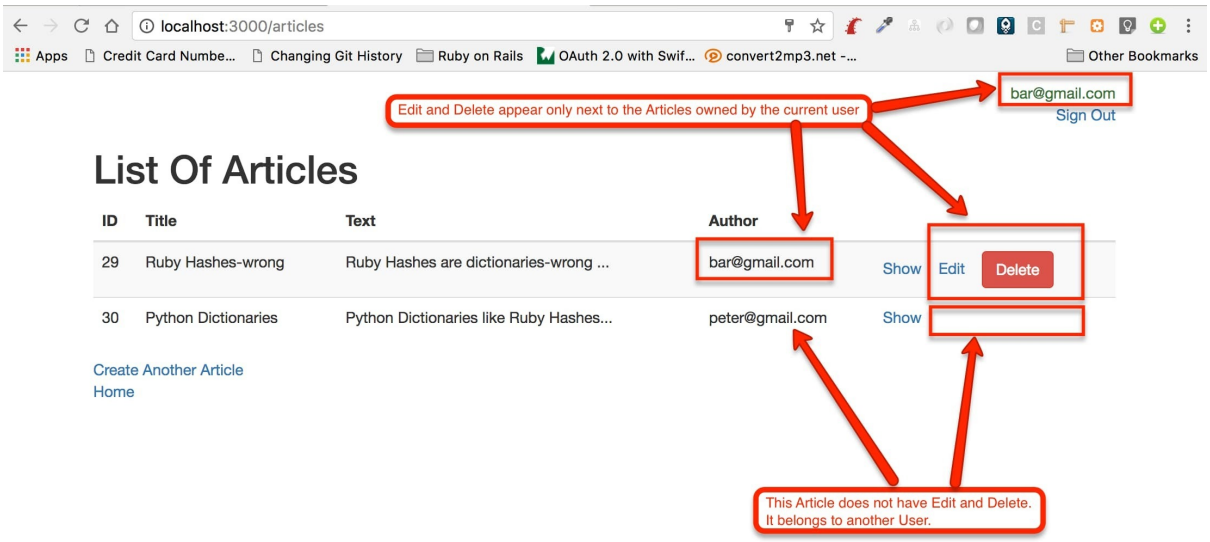
You can see lines 22 and 24. These are the lines that we did a change on. We have added the `current_user.can_edit_article?(article)` and the `current_user.can_delete_article?(article)` respectively.

According to this, the `User` model needs to respond to these two `can_....` questions. Let's see how we could implement this. Here is the `User` model updated.

```ruby
# File app/models/user.rb
#
class User < ApplicationRecord
  validates :email, presence: true, uniqueness: { case_sensitive: false }
  has_secure_password

  before_create :generate_email_confirmation_token

  def email_confirmed?
    !email_confirmation_token.present?
  end

  def confirm_email_address
    self.email_confirmation_token = nil
    save
  end

  def can_edit_article?(article)
    article.user.id == id
  end

  def can_delete_article?(article)
    article.user.id == id
  end

  private

  def generate_email_confirmation_token
    self.email_confirmation_token = SecureRandom.urlsafe_base64
  end
end
```

(the above code snippet online)

Cool! Now, try to load the List of Articles page. You will see something like this:



Edit and Delete Appear Next To Own Articles

We can hide the links on the Article show view too (`app/views/articles/show.html.haml`):

This is the new version of this file that hides the links that are not applicable:

```
- # File: app/views/articles/show.html.haml
- #
%div
  %strong ID:
  = @article.id
%div
  %strong Title:
  = @article.title
%hr/
%div
  %strong Text:
%div
  = @article.text
- if current_user && current_user.can_edit_article?(@article)
  %div
    = link_to 'Edit Article', edit_article_path(@article)
- if current_user
  %div
    = link_to 'Create New Article', new_article_path
%div
  = link_to 'List of Articles', articles_path
%div
  = link_to 'Home', root_path
```

(the above code snippet online)

Look at line 14. You can see that we have added an extra condition for the `Edit` link to appear. The `current_user` needs to be able to edit the Article at hand.

Drying Out Code

We have implemented authorization for the Articles management, but we have, unfortunately, spread the logic of whether a user can update an Article or not. See the `user.rb` model and the `authorize_user` method inside the `ArticlesController`. We encounter, many times, the condition that the Article `user.id` needs to be the same like the `id` of the current user.

First, we need to understand that we want to know who can edit/update/delete an Article and who can only view it. Hence, we can have two privileges:

1. write
2. read-only

In (1), write, privilege, the user can edit/update/delete an Article. In (2), read-only, a user can only view an Article.

Having said that, then `User` model can, now, rely only on the method `can_write_article?`. This is the new version of it:

```
# File app/models/user.rb
#
class User < ApplicationRecord
  validates :email, presence: true, uniqueness: { case_sensitive: false }
  has_secure_password

  before_create :generate_email_confirmation_token

  def email_confirmed?
    !email_confirmation_token.present?
  end

  def confirm_email_address
    self.email_confirmation_token = nil
    save
  end

  def can_write_article?(article)
    article.user.id == id
  end

  private

```

```
  def generate_email_confirmation_token
    self.email_confirmation_token = SecureRandom.urlsafe_base64
  end
end
```

(the above code snippet online)

The methods `can_edit_article?` and `can_delete_article?` are no longer needed.

We will use this `can_write_article?()` method both in the `ArticlesController` and in the views.

Here is the `def authorize_user` method of the `ArticlesController`:

```
  def authorize_user
    unless current_user.can_write_article?(@article)
      flash[:warning] = 'Your are not allowed to act on this Article!'
      redirect_to root_url
    end
  end
```

(the above code snippet online)

You can see how we now use the `current_user.can_write_article?(@article)`.

Let's change also the way we hide the Edit and Delete actions in `app/views/articles/index.html.haml` file:

```
- # File: app/views/articles/index.html.haml
- #
%h1 List Of Articles
%table.table.table-striped
  %thead
    %tr
      %th ID
      %th Title
      %th Text
      %th Author
      %th
  %tbody
    - @articles.each do |article|
      %tr
        %td= article.id
        %td= article.title
        %td= article.text
        %td= article.user.email
        %td
          = link_to 'Show', article_path(article)
          - if current_user && current_user.can_write_article?(article)
```

```
22                
23              = link_to 'Edit', edit_article_path(article)
24                
25              = link_to 'Delete', article_path(article), method: :delete, class: 'b\
26 tn btn-danger', data: {confirm: 'Are you sure that you want to delete this Articl\
27 e?'}
28 - if current_user
29   %div
30     = link_to 'Create Another Article', new_article_path
31 %div
32   = link_to 'Home', root_path
```

(the above code snippet online)

Look on lines 21 till 25. We now put the links inside the block that makes sure that we have a signed in user and that the particular user can write the article at hand.

Finally, do the appropriate change in the file `app/views/articles/show.html.haml`:

```
1  - # File: app/views/articles/show.html.haml
2  - #
3  %div
4    %strong ID:
5    = @article.id
6  %div
7    %strong Title:
8    = @article.title
9  %hr/
10 %div
11   %strong Text:
12 %div
13   = @article.text
14 - if current_user && current_user.can_write_article?(@article)
15   %div
16     = link_to 'Edit Article', edit_article_path(@article)
17 - if current_user
18   %div
19     = link_to 'Create New Article', new_article_path
20 %div
21   = link_to 'List of Articles', articles_path
22 %div
23   = link_to 'Home', root_path
```

(the above code snippet online)

Other Authorization Gems

We have implemented authorization with our own Ruby code. However, you may want to rely on a publicly available gem that is especially designed for that purpose. Such a gem is cancancan, for example. It is very popular and resource-based. Now that you have become familiar with what authorization is, you may want to work with one of these libraries, that offer a lot of features and have many capabilities.

Closing Note

We saw the authorization which is the code that checks whether a user is allowed to do specific things. Authentication and authorization go hand-in-hand and as soon as you master these ideas you will be able to work with more advanced authentication and authorization libraries like devise and cancancan.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

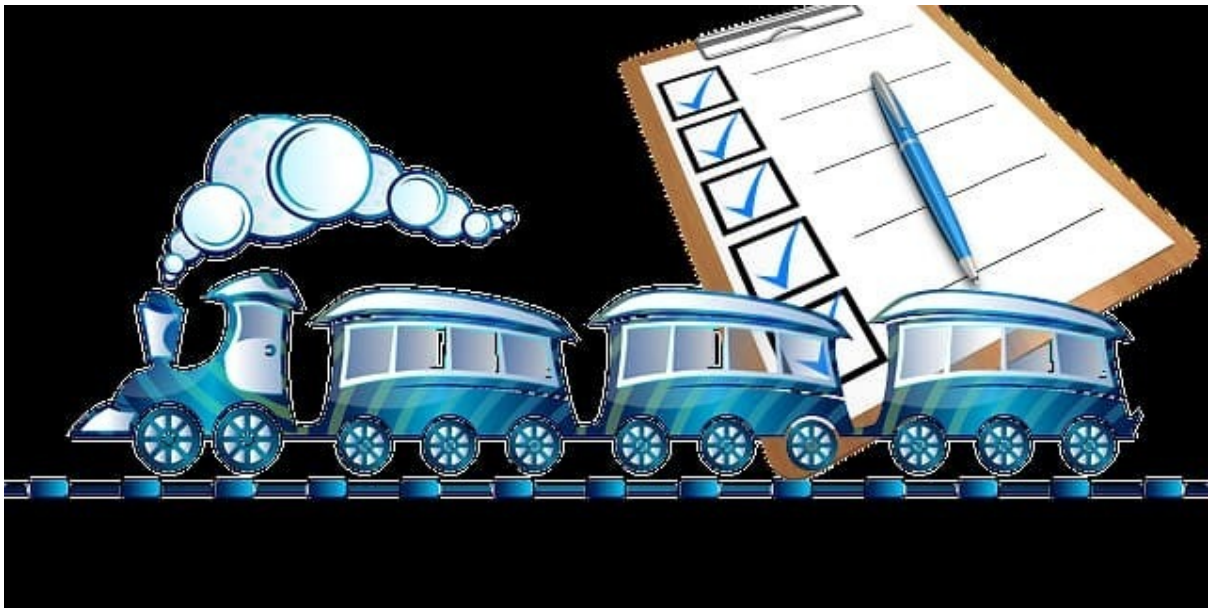
Make sure that you add to your `blog` application all the authorization features that are described in the chapter content.

Push your code to your Github repository.

Deploy to Heroku.

6 - Testing Rails Applications

Summary



Rails Testing

Ruby on Rails projects have the test automation built into their hearts. Actually, it is built into Ruby on Rails framework itself. This chapter is a must study, difficult chapter, that teaches you the main testing techniques that you may encounter in your professional life with Ruby on Rails. You will learn how to implement model tests, integration tests and system tests. `RSpec` and `Cucumber` are also added to the scene so that you know how these tools are integrated with RoR. Again, this is a chapter that you need to master very well in order to be competent Web developer.

Learning Goals

1. Learn about the test folder structure.
2. Learn about the different environments in Rails.
3. Learn how to set up your test environment to work well with the emails and the URLs that might be included.
4. Learn how to tell which environment your code has been launched in.
5. Learn how to start Rails console in an environment different from `development`.
6. Learn how to start Rails server in different environments.
7. Learn how Heroku sets the environment for your production application.
8. Learn how to run rake tasks in different environments.
9. Learn how the test environment has its own test database.
10. Learn about the class hierarchy that is related to tests.
11. Learn how you can write tests on Models.
12. Learn about the `test_helper.rb` and what is its role.

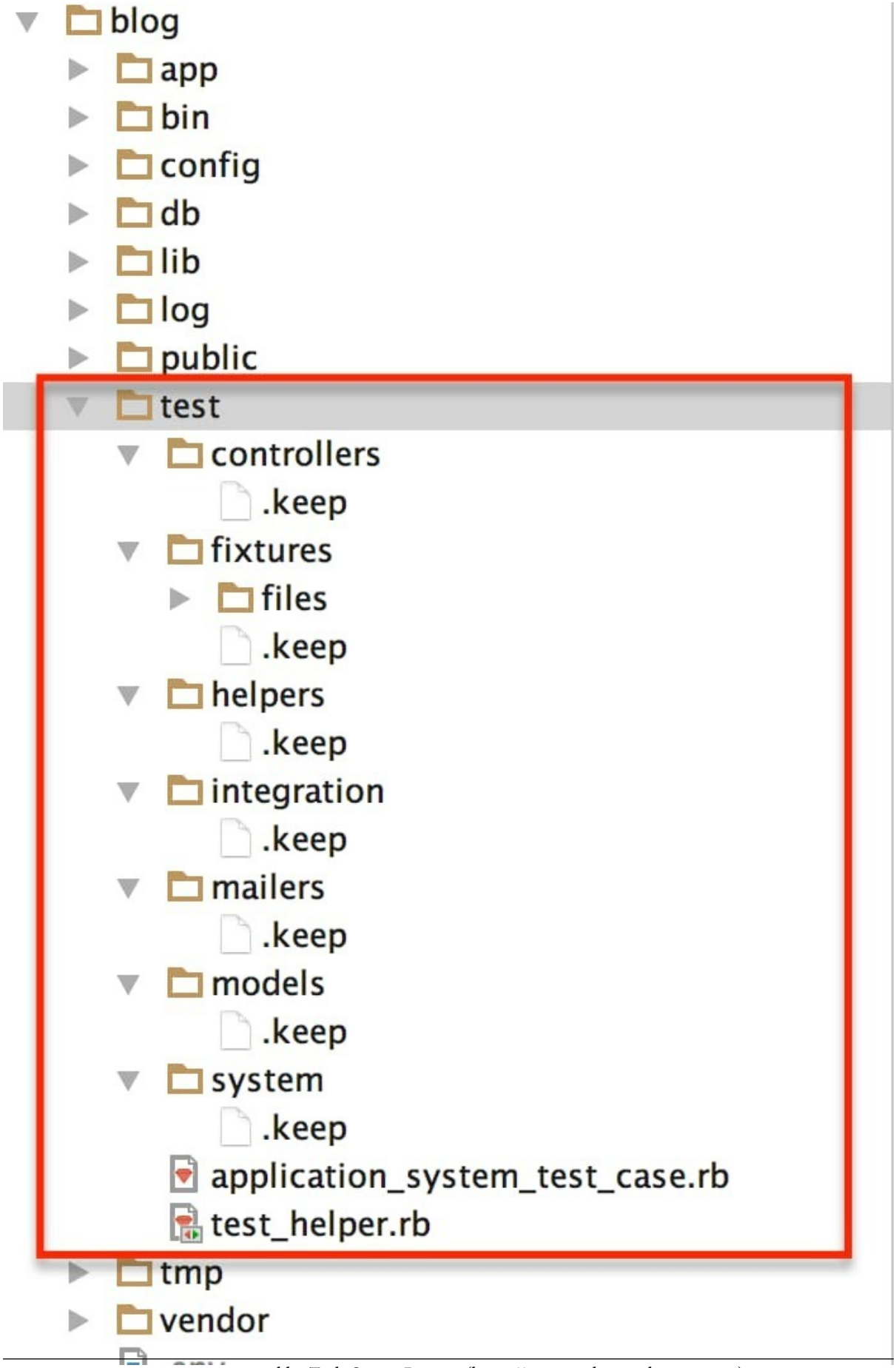
13. Learn about the `ActiveSupport::TestCase` assertions.
14. Learn how you can run your test suite.
15. Learn about the fixtures.
16. Learn about system tests and Capybara.
17. Learn how to run the system tests.
18. Learn how to install chromedriver.
19. Learn how you can use Capybara to write your system tests.
20. Learn how to get a screenshot of your failed test when running system tests.
21. Learn about the headless `:webkit` driver.
22. Learn about the integration tests.
23. Learn about controller tests.
24. Learn how to write uni tests for your Mailers.
25. Learn how to integrate RSpec with your Ruby on Rails Web Application.
26. Learn how to use fixtures in your RSpec specifications.
27. Learn how to write model, system and integration tests with RSpec.
28. Learn how you can integrate Cucumber with your Ruby on Rails project.
29. Learn how to write Model features using Cucumber.
30. Learn how to write System features using Cucumber.
31. Learn about factories and Factory Girl.
32. Learn how to use factories to eliminate some code duplication from your tests.

Introduction

Rails comes with automated testing tools already in and it is its philosophy to cover your application features with automated tests so that you can make sure you don't break your application every time you introduce a new feature.

Test Folder Tree

When you create a new Rails application, one of the folders that it is created is the `test` folder. Here is a picture of its tree:



content is owned by Tech Career Booster (https://www.techcareerbooster.com)

Test Folder Tree

Rails suggests folders for you to put / write your automated tests in. For example, for the Model tests, it suggests you to write them inside the folder `models`.

We will write tests for each one of these folders in this chapter and you will understand better how they are supposed to be used.

Test Environment

Before we proceed with the writing of some tests for our `blog` application, I would like to make sure that you understand the idea of the *environment*.

Every new Rails application comes with support for three different environments, by default.

1. `development`
2. `test`
3. `production`

The different environments have common configuration option values, which are specified inside the file `config/environment.rb`. But, they also have environment-specific configuration option values, which are specified inside the file which is specific per environment:

1. For `development` environment, the file is the `config/environments/development.rb`.
2. For `test` environment, the file is the `config/environments/test.rb`.
3. For `production` environment, the file is `config/environments/production.rb`.

For example, a notable difference between `development` and `test` environment is that in the `test` environment the emails are not actually delivered/sent out to an SMTP server. The `delivery_method` is set to `:test`. Look for the `config.action_mailer.delivery_method = :test` line inside your `config/environments/test.rb` file.

However, this is the point at which you need to specify the `default_url_options` for the `test` environment too. Otherwise, your tests, that exercise mailing delivery too, might fail. So, go to your `config/environments/test.rb` and add the following:

```
config.action_mailer.default_url_options = {protocol: 'http', host: 'localhost:30\
00'}
```

(the above code snippet online)

Starting Rails Console In Different Environments

When you start `rails console`, by default, it starts in `development` environment. This can be confirmed in the following ways as depicted below (try that on your machine):

```
blog $ bin/rails console
Loading development environment (Rails 5.1.1)
2.3.3 :001 > ENV['RAILS_ENV']
 => "development"
2.3.3 :002 > Rails.env
 => "development"
2.3.3 :003 > Rails.env.development?
 => true
2.3.3 :004 > Rails.env.test?
 => false
2.3.3 :005 > Rails.env.production?
 => false
2.3.3 :006 >
```

(the above code snippet online)

1. You can see that the `Loading ....` statement at start mentions that the environment being loaded is the `development` one.
2. The environment variable `RAILS_ENV` is set to `development`.
3. The method `Rails.env` returns `development`.
4. The method `Rails.env.development?` returns `true`.
5. The method `Rails.env.test?` returns `false`.
6. The method `Rails.env.production?` returns `false`.

How can we start Rails console in another environment? One way you can do that is by setting the `RAILS_ENV` environment variable to have the value of the environment you want Rails console to start in.

Exit the previous console and then start it again as follows:

```
blog $ RAILS_ENV=test rails console
Loading test environment (Rails 5.1.1)
2.3.3 :001 > ENV['RAILS_ENV']
 => "test"
2.3.3 :002 > Rails.env
 => "test"
2.3.3 :003 > Rails.env.development?
 => false
2.3.3 :004 > Rails.env.test?
 => true
2.3.3 :005 > Rails.env.production?
 => false
2.3.3 :006 >
```

(the above code snippet online)

Another way you can start Rails console in another environment is to specify it as command line argument. Exit the rails console that you are in and then start it again with:

```
blog $ bin/rails console test
Loading test environment (Rails 5.1.1)
2.3.3 :001 >
```

(the above code snippet online)

The `bin/rails console test` starts Rails console in `test` environment.

Starting Rails Server In Different Environments

We know that Rails server can be started with `bin/rails server`. This, again, starts Rails server in `development` environment. We can start the Rails server in another environment by first setting the `RAILS_ENV` environment variable:

```
blog $ RAILS_ENV=test bin/rails server
=> Booting Puma
=> Rails 5.1.1 application starting in test on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 5, max threads: 5
* Environment: test
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

(the above code snippet online)

Do you see the `Environment: test` being printed? This confirms that the Rails server has started in the environment specified by the `RAILS_ENV` environment variable.

Stop your Rails server and then try to start it with the following command:

```
blog $ bin/rails server --environment=test
=> Booting Puma
=> Rails 5.1.1 application starting in test on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 5, max threads: 5
* Environment: test
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

(the above code snippet online)

The `bin/rails server --environment=test` is another way to start your Rails server in `test` environment.

Production on Heroku

I guess that, by now, environments are pretty much clear to you. Final note here is that when you deploy your application to Heroku, Heroku has `RAILS_ENV` set to `production` by default. Hence, your Web application will be initialized using this environment, i.e. the `production` environment.

You can confirm that by running `heroku config --app panosm-blog | grep 'RAILS_ENV'` on your command line (replacing `panosm-blog` with the name of your blog application):

```
blog $ heroku config --app panosm-blog | grep 'RAILS_ENV'
RAILS_ENV:                production
blog $
```

(the above code snippet online)

`rake` Tasks

Note that all the `rake` tasks run in `development` environment by default. If you want to run a rake task against another environment, the suggested way is that you set the `RAILS_ENV` environment variable accordingly.

E.g. `RAILS_ENV=production rake db:migrate` would run migrations against the `production` environment.

Test Database

Each environment has its own database. The access details for the `test` database are found inside the `config/database.yml` file, in the `test` section. Having a different test database to run our tests against is very useful. We can do anything that we like without worrying about destroying development or production data.

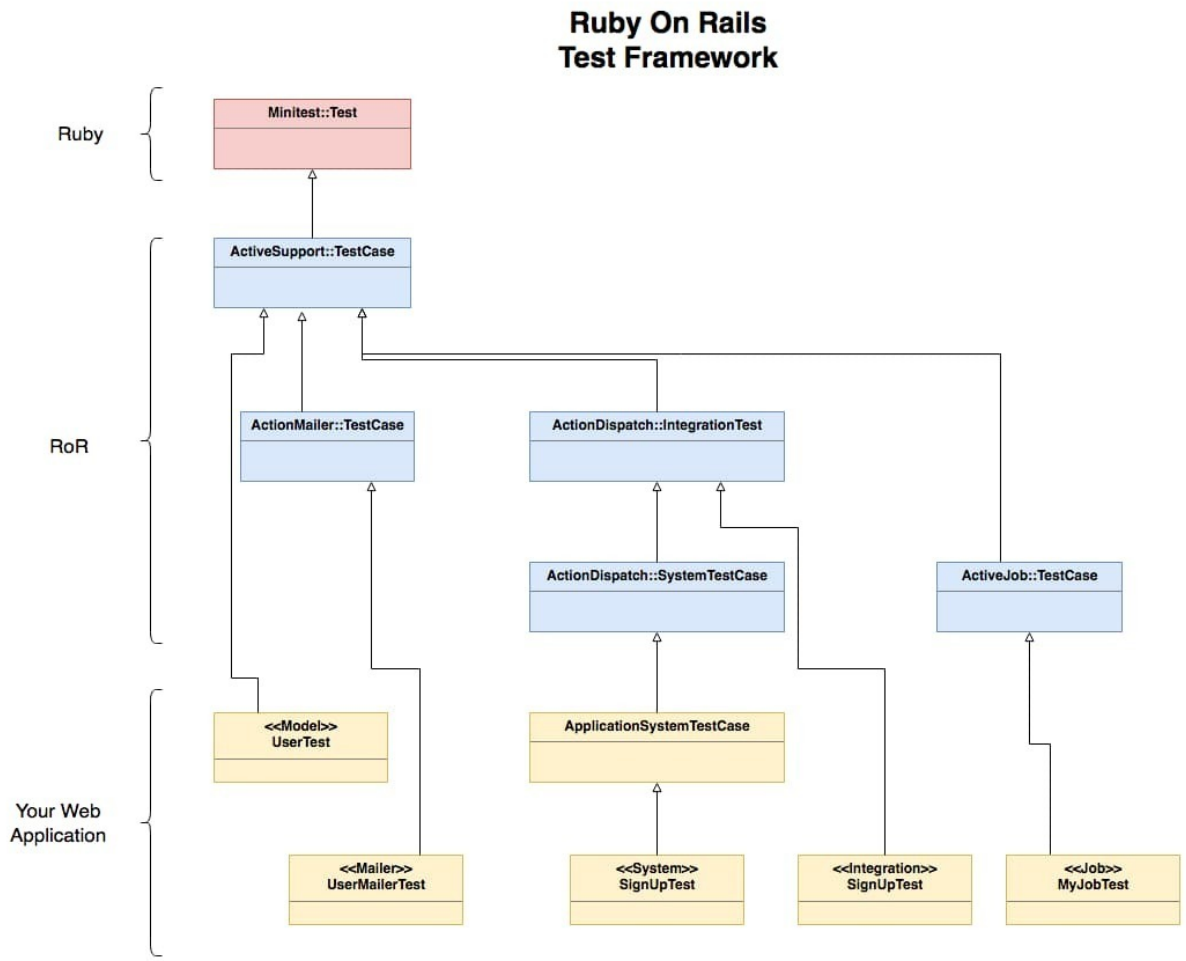
Note that when your tests start to run, the `test/test_helper.rb` file includes logic to load the file `db/schema.rb` and prepare your test database. More about this when we will talk about schema migrations.

Ruby on Rails Test Framework Class Hierarchy

In the following paragraphs we will write tests for:

1. Models
2. Integration Tests
3. System Tests
4. Mailer Tests

All the tests should be written as classes that derive from specific superclasses depending on the above cases. The following diagram will help you.



Ruby On Rails Test Classes Hierarchy

Object of Test	Your Test Class Derives From
Models	`ActiveSupport::TestCase`
Integration Tests	`ActionDispatch::IntegrationTest`
System Tests	`ActionDispatch::SystemTestCase` or even better `ApplicationSystemTestCase`
Mailer Tests	`ActionMailer::TestCase`

Note that in this chapter we will not talk about Jobs and how we test them. It is a subject of an upcoming chapter.

Also, note that there are a couple of other test-related classes that Rails comes with, but their usage is deprecated.

Testing Models

Let's start by covering with tests one of our most important models, the `User`.

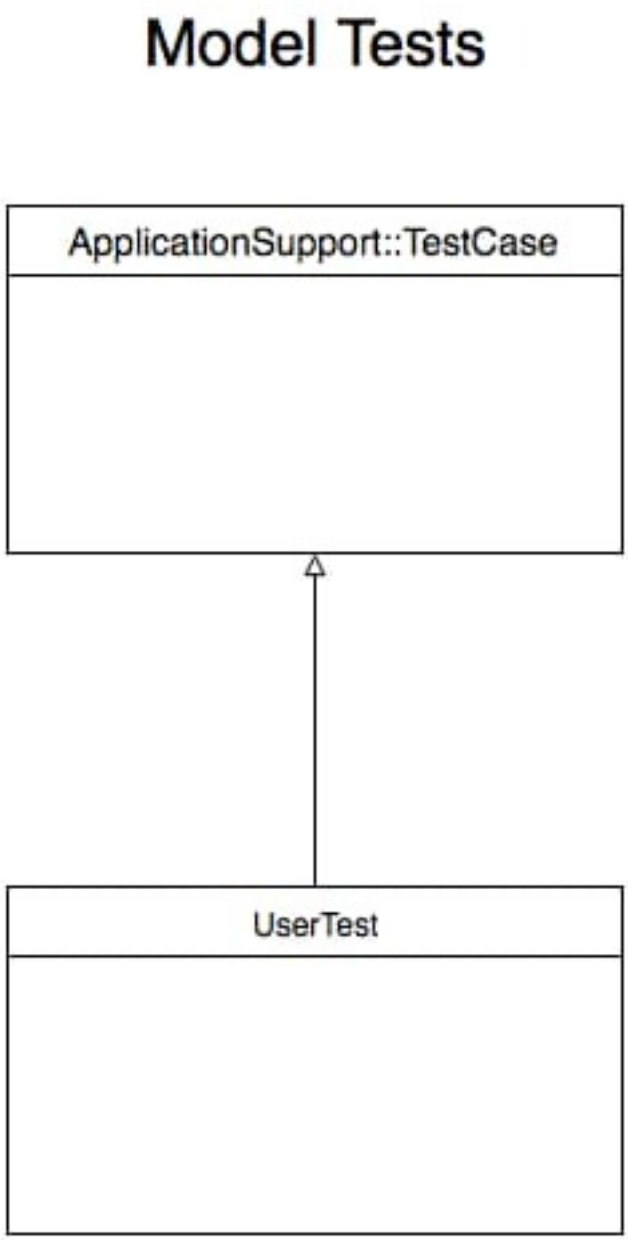
You need to create a file named `user_test.rb` inside `test/models` folder. Start with something like this:

```
1 # File: test/models/user_test.rb
2 #
3 require 'test_helper'
4
5 class UserTest < ActiveSupport::TestCase
6   test 'it should be true' do
7     assert(true)
8   end
9 end
```

(the above code snippet online)

All of our test files are going to require `test_helper`. Hence in `test_helper.rb` we will be defining things that are necessary to be available to all of our test files.

The line 5 defines the class `UserTest` that derives from the `ActiveSupport::TestCase`. It will have at its disposal all the methods that the `ActiveSupport::TestCase` class offers.



Model Tests - Derive From ActiveSupport::TestCase

But `ActiveSupport::TestCase` derives from `Minitest::Test`. Which means all the functionality offered by `Minitest` is already in.

> **Important**: This is the list of all the assertions that `ActiveSupport::TestCase` offers and that you can use on top of the `Minitest` assertions

Inside the `UserTest` class we call the class-level method `test` to define a test scenario. We give the name of the test, e.g. `'it should be true'`, and inside the block `do...end` sent to this call we give the code that constitutes the test code.

Lines 6 to 8 above have a dummy test, that we will use to make sure that we can run our tests.

Note that Rails test commands, like `assert`, are coming from `Minitest`, the testing framework that we have talked about in the chapter "Section: Test Automation, TDD and BDD, Chapter: Minitest". However, the set of available assertions has been enriched to cover for the Web development and Rails components. We will learn about these later on.

Running the Test Suite

The easiest way for you to run your test suite is by calling Rails test runner: `bin/rails test`:

```
blog $ bin/rails test
Running via Spring preloader in process 69402
Run options: --seed 4434

# Running:

.

Finished in 0.046741s, 21.3945 runs/s, 64.1836 assertions/s.
1 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

(the above code snippet online)

This proves that we can run our tests successfully. The default `rake` task is the task that invokes the running of your test suite.

Tests for `User` Model

The first test that we can write for the `User` model is the following:

```ruby
# File: test/models/user_test.rb
#
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test 'should not save user without email' do
    user = User.new(password: '12345678')
    assert_not user.save
  end
end
```

(the above code snippet online)

The above makes sure that we cannot save a User without email.

Run the suite:

```
blog $ bin/rails test
Running via Spring preloader in process 69714
Run options: --seed 8338

# Running:

.

Finished in 0.035187s, 28.4192 runs/s, 28.4192 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

(the above code snippet online)

Let's see when this test fails. Change the `User` model and remove the presence validation for email:

```
...
  # validates :email, presence: true, uniqueness: { case_sensitive: false }
  validates :email, uniqueness: { case_sensitive: false }
...
```

(the above code snippet online)

We have put the proper `validates` statement in comments and we have added a temporary one that does not include the presence validation.

Let's run the suite again:

```
Run options: --seed 49868

# Running:

E

Error:
UserTest#test_should_not_save_user_without_email:
ActiveRecord::NotNullViolation: PG::NotNullViolation: ERROR:  null value in colum\
n "email" violates not-null constraint
DETAIL:  Failing row contains (1, null, $2a$04$rOJlK19v9.a3yef8rI7rR.ao1DWD5jXNnP\
MSJQlYgJB8x3m8I5LKO, 2017-06-20 08:20:39.527772, 2017-06-20 08:20:39.527772, V_il\
BP3Z3H5JVZCLBifZPw).
: INSERT INTO "users" ("password_digest", "created_at", "updated_at", "email_conf\
irmation_token") VALUES ($1, $2, $3, $4) RETURNING "id"
    test/models/user_test.rb:8:in `block in <class:UserTest>'


bin/rails test test/models/user_test.rb:6

```

```
Finished in 0.032598s, 30.6769 runs/s, 0.0000 assertions/s.
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

(the above code snippet online)

We can see that the test now fails.

Let's put the presence validation back in again. Then run the suite again. It should succeed.

Now, we will try to make the test more specific to what it is trying to test.

```
1  # File: test/models/user_test.rb
2  #
3  require 'test_helper'
4
5  class UserTest < ActiveSupport::TestCase
6    test 'should not save user without email' do
7      user = User.new(password: '12345678')
8
9      assert_not user.save
10     assert user.errors[:email].include?("can't be blank")
11   end
12 end
```

(the above code snippet online)

On line 10, we also make sure that the save failure is due to the fact that the email was blank. We do that by checking the errors that are now attached to the `:email` attribute of the `user` instance. When the presence validation fails, then Rails attaches the error `"can't be blank"` to the attribute that is not present, when it had to.

Now, run the test suite again. You will see that it runs without any problems.

Note that the assertion `assert user.errors[:email].include?("can't be blank")` can actually be written with the help of the assertion `assert_includes`:

```
# File: test/models/user_test.rb
#
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test 'should not save user without email' do
    user = User.new(password: '12345678')

    assert_not user.save
    assert_includes user.errors[:email], "can't be blank"
  end
end
```

(the above code snippet online)

Let's add one more test to the `User` model:

```ruby
# File: test/models/user_test.rb
#
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test 'should not save user without email' do
    user = User.new(password: '12345678')

    assert_not user.save
    assert_includes user.errors[:email], "can't be blank"
  end

  test 'generates email confirmation token' do
    # setup
    user = User.new(email: 'foo@bar.com', password: '12345678')

    assert user.email_confirmation_token.blank?

    # fire
    user.save!

    # check results
    assert_not user.email_confirmation_token.blank?
  end
end
```

(the above code snippet online)

We have added line 13 to 24 that make sure that when a new User is created, it is assigned an email confirmation token. You can see that I have clearly separated the test into three phases. The setup phase, the fire phase and the check results phase. This is sometimes quite useful to do because it improves readability of your test code.

Run the test suite again. You will see two dots representing the tests above:

```
blog $ bin/rails test

Run options: --seed 25910

# Running:

..

Finished in 0.042646s, 46.8976 runs/s, 117.2439 assertions/s.
2 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

(the above code snippet online)

There are many other tests that we can write at the `User` model level. But we will leave those for the task of this chapter.

Fixtures

Rails allows you to define some sample data using what it is called *fixtures*. The fixtures are being loaded before the tests run.

The definition of fixtures is done using YAML files. Usually, there is one YAML file per model.

Let's create the file `users.yml` inside the `test/fixtures` folder:

```
# File: test/fixtures/users.yml
#
peter:
  email: peter@gmail.com
  password_digest: <%= SecureRandom.urlsafe_base64 %>

mary:
  email: mary@gmail.com
  password_digest: <%= SecureRandom.urlsafe_base64 %>
```

(the above code snippet online)

The above `users.yml` fixtures file defines two users. One with identifier `peter` and another one with identifier `mary`.

You can use statements like `users(:peter)` to get access to the `Peter` object. Or `users(:mary)` to get access to the `Mary` object. These users are created before your tests start to run.

Note also how we use ERB to run Ruby code and help us set the values for `password_digest`. The parser that parses fixtures YAML files first sends the file to the ERB parser. This is very useful. For example, you will need ERB inside the fixtures YAML file if you were to create 1000 users in your database.

System Testing

We now move to the systems tests. The system tests are testing your application as the User perceives it. In other words, they test the User experience. System tests are using, internally, the capybara gem. Capybara offers an API that allows you to browse the Web interface of your application using Ruby.

Testing Sign Up

As an example of system test that uses Capybara, we are going to write the tests for the sign up feature.

```ruby
# File: test/system/sign_up_test.rb
#
require 'application_system_test_case'

class SignUpTest < ApplicationSystemTestCase
  test 'sign up happy path' do
    visit root_url

    #.... more things to be written here ....
  end
end
```

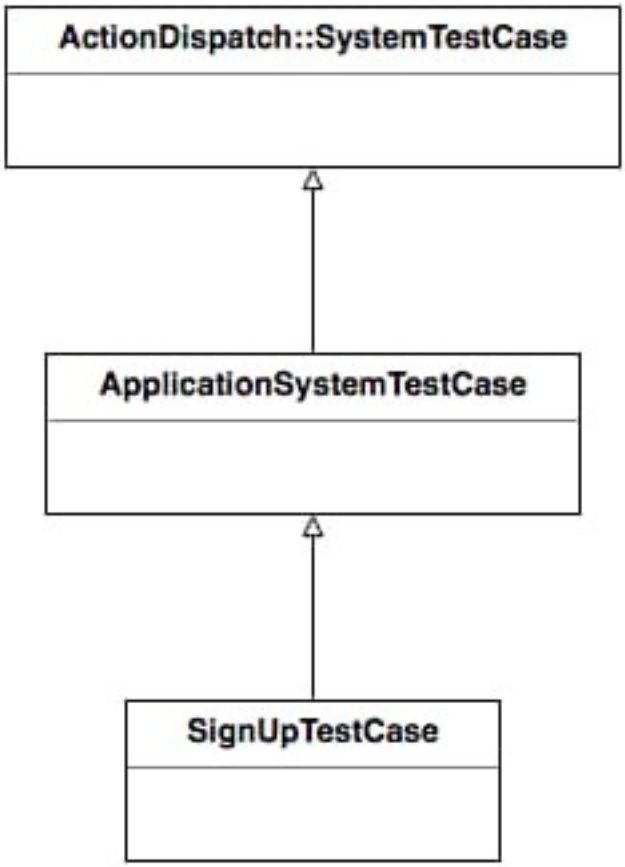
(the above code snippet online)

The above is the file `test/system/sign_up_test.rb` and requires `test/application_system_-test_case.rb` with the proper defaults defined there. If you see the file `test/application_-system_test_case.rb` you will see that it uses, by default, the `selenium` driver, which launches a real `chrome` window with size `1400 x 1400`:

```ruby
# File: test/application_system_test_case.rb
#
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  driven_by :selenium, using: :chrome, screen_size: [1400, 1400]
end
```

(the above code snippet online)

Note that all the system test cases, like the `SignUpTest`, need to have as superclass the `ApplicationSystemTestCase`, which derives from `ActionDispatch::SystemTestCase`.



System Test Cases Derive From ActionDispatch::SystemTestCase

> **Important:** Actually, `ActionDispatch::SystemTestCase` derives from `ActionDispatch::IntegrationTest` which derives from `ActiveSupport::TestCase`. We will see the `ActionDispatch::IntegrationTest` later on, when we will discuss the Integration Tests. But, you need to know that an `ActionDispatch::SystemTestCase` derives the functionality that the `ActionDispatch::IntegrationTest` has too.

Try to run the Rails test runner as follows. Note that you have to specify the `test:system` and not just test because the simple test runner doesn't execute the systems tests:

```
blog $ bin/rails test:system
Run options: --seed 6717

# Running:

E

Error:
SignUpTest#test_sign_up_happy_path:
Selenium::WebDriver::Error::WebDriverError:  Unable to find chromedriver. Please \
download the server from http://chromedriver.storage.googleapis.com/index.html an\
```

```
d place it somewhere on your PATH. More info at https://github.com/SeleniumHQ/sel\
enium/wiki/ChromeDriver.

    test/system/sign_up_test.rb:7:in `block in <class:SignUpTest>'

Error:
SignUpTest#test_sign_up_happy_path:
Selenium::WebDriver::Error::WebDriverError:  Unable to find chromedriver. Please \
download the server from http://chromedriver.storage.googleapis.com/index.html an\
d place it somewhere on your PATH. More info at https://github.com/SeleniumHQ/sel\
enium/wiki/ChromeDriver.




bin/rails test test/system/sign_up_test.rb:6

Finished in 0.164582s, 6.0760 runs/s, 0.0000 assertions/s.
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

(the above code snippet online)

You might see an error like the above, which tells you that `chromedriver` was not found. Also, it gives you instructions how to download and install it.

Personally, on my Mac OS X, I have done `brew install chromedriver` and the latest version was installed.

Then I tried to run the test again:

```
blog $ bin/rails test:system
Run options: --seed 15111

# Running:

Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 0, max threads: 1
* Environment: test
* Listening on tcp://0.0.0.0:57644
Use Ctrl-C to stop
.

Finished in 2.536624s, 0.3942 runs/s, 0.0000 assertions/s.
1 runs, 0 assertions, 0 failures, 0 errors, 0 skips
blog $
```

(the above code snippet online)

And the test ran successfully.

Let's now proceed to amend the Sign Up test to have the form of Sign Up filled in and test that we see what it has to be seen. This is the new version of the Sign Up system test:

```
1  # File: test/system/sign_up_test.rb
2  #
3  require 'application_system_test_case'
4
5  class SignUpTest < ApplicationSystemTestCase
6    test 'sign up happy path' do
7      email = "#{SecureRandom.hex}@mailinator.com"
8      test_password = '12345678'
9
10     visit root_url
11
12     click_link 'Sign Up'
13
14     fill_in 'Email', with: email
15     fill_in 'Password', with: test_password
16     fill_in 'Password confirmation', with: test_password
17
18     click_button 'Sign Up'
19
20     assert_selector '.alert.alert-success', text: 'You have successfully signed u\
21 p! Please, confirm your email address. We just sent you out an email with a link \
22 to do so!'
23   end
24 end
```

(the above code snippet online)

First, we visit the root page of our Web application (line 10). Then we click the `Sign Up` link and we visit the Sign Up page (line 12).

Then we fill in the Sign Up form (lines 14 to 16) and we submit it (line 18).

Finally, we make sure that we see the success alert with the correct content.

The Ruby statements that we use in this test are basically Capybara methods more of which you can find here.

See also the very generic assertion command `assert_selector`. This makes sure that the HTML page rendered includes the HTML tags specified.

Run the above test with `bin/rails test:system`:

```
blog $ bin/rails test:system
Run options: --seed 7577

# Running:

Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 0, max threads: 1
* Environment: test
* Listening on tcp://0.0.0.0:64575
Use Ctrl-C to stop
.

Finished in 3.598572s, 0.2779 runs/s, 0.2779 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
blog $
```

(the above code snippet online)

Cool! As you can see the test has run successfully.

Do you want to make this test go red? Just to see it failing and then going green again?

Let's change our `SignUpController`. We will introduce a bug. Go to the method `def user_params` and change it to be:

```
  def user_params
    params.require(:user).permit(:email)
  end
```

(the above code snippet online)

We have removed the `:password` and `:password_confirmation` from the list of permitted attributes.

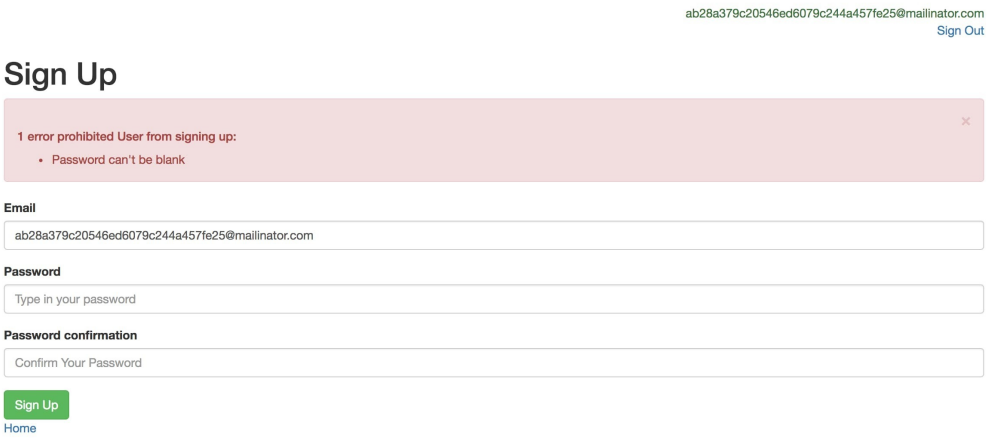
Now, run the test again:

```
blog $ bin/rails test:system
Run options: --seed 6165

# Running:

Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 0, max threads: 1
* Environment: test
* Listening on tcp://0.0.0.0:51135
Use Ctrl-C to stop
[Screenshot]: tmp/screenshots/failures_test_sign_up_happy_path.jpg
```

```

F

Failure:
SignUpTest#test_sign_up_happy_path [/Users/panayotismatsinopoulos/Documents/techc\
areerbooster_course_content/courses/01-full-stack-web-developer/sections/15-ruby-\
on-rails/chapters/06-testing-rails-applications/exercises-and-code/blog/test/syst\
em/sign_up_test.rb:20]:
expected to find css ".alert.alert-success" with text "You have successfully sign\
ed up! Please, confirm your email address. We just sent you out an email with a l\
ink to do so!" but there were no matches


bin/rails test test/system/sign_up_test.rb:6

Finished in 6.020758s, 0.1661 runs/s, 0.1661 assertions/s.
1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
```

(the above code snippet online)

You can see that the test fails. When a test fails, you can also find a screenshot of your application at the point it failed, stored inside the folder `tmp/screenshots`. This is what I've got when the test failed on my machine:



Screenshot of Failure

Let's revert the change to bring the code back to its correct state. And run the test again. You will see it succeed.

Headless Driver

`:selenium` is a driver that starts a full browser to run your tests. This might slow down your tests. For that reason, you may want to try a headless driver. One such famous driver is `:webkit` which requires the `capybara-webkit` gem to be installed.

Please, go ahead and amend your `Gemfile`, in the `:test` group, to include the `gem 'capybara-webkit'` statement. Then do `bundle` to install `capybara-webkit`.

Then, go to your `test/application_system_test_case.rb` file and change it so that it works with `:webkit` driver:

```ruby
# File: test/application_system_test_case.rb
#
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  driven_by :webkit
end
```

(the above code snippet online)

Now, run the test again. This is what I've got on my end:

```
blog $ bin/rails test:system
Run options: --seed 33751

# Running:

Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 0, max threads: 1
* Environment: test
* Listening on tcp://0.0.0.0:64503
Use Ctrl-C to stop
.

Finished in 2.038362s, 0.4906 runs/s, 0.4906 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

(the above code snippet online)

If you read the time spent to finish the test, you will see that it is much less than when we used the `:selenium` driver.

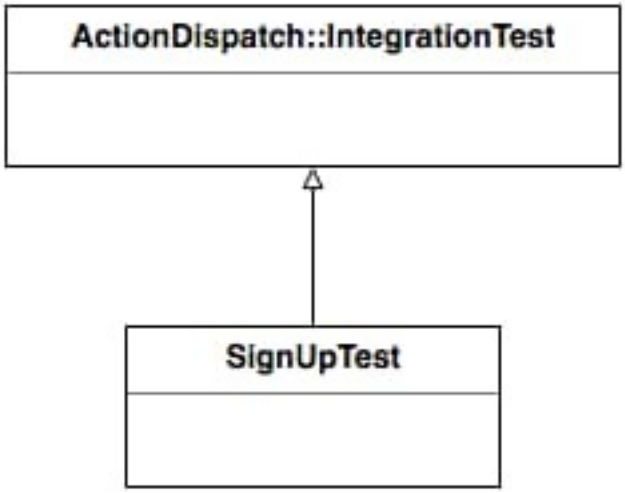
Integration Testing

Although the system tests use Capybara and simulate the interaction of the user with the Web UI, integration tests do not assume a browser, but they assume a REST client sending REST requests to a REST server, i.e. your Web applications.

Also, integration tests can simulate multiple sessions running in parallel, or in sequence, during the same test case, which is an important difference to system testing.

However, you need to know that system testing is much more close to testing the actual user interaction with our Web application. Integration testing does not do that.

Although system test cases need to derive from `ApplicationSystemTestCase`, which derives from `ActionDispatch::SystemTestCase`, integration tests need to derive from `ActionDispatch::IntegrationTest`.



Integration Tests Derive From ActionDispatch::IntegrationTest

Sign Up Integration Test

Let's now write the sign up feature test in its integration form.

We need to create the file `sign_up_test.rb` inside the `test/integration` folder as follows:

```
# File: test/integration/sign_up_test.rb
#
require 'test_helper'

class SignUpTest < ActionDispatch::IntegrationTest
  test 'sign up happy path' do
    email = "#{SecureRandom.hex}@mailinator.com"
    test_password = '12345678'

    # 1. send a GET request to the sign up path
    get sign_up_path
    assert_response :success

    assert_select 'input[name="user[email]"]'
    assert_select 'input[name="user[password]"]'
    assert_select 'input[name="user[password_confirmation]"]'

    # 2. send a POST request to the sign up path, and follow the redirect
    post sign_up_path, params: {user: {email: email, password: test_password, pas\
sword_confirmation: test_password}}
    assert_response :redirect

    follow_redirect!
    assert_response :success
```

```
25
26      # 3. check now what is inside the HTML content
27      assert_select '.alert.alert-success', /You have successfully signed up! Pleas\
28 e, confirm your email address\. We just sent you out an email with a link to do s\
29 o!/
30    end
31  end
```

(the above code snippet online)

Here it is what is going on the code above:

1. On line 5, we create the class `SignUpTest` and make it derive from `ActionDispatch::IntegrationTest`.
2. Then on line 6, we define the test.
3. We use the method `get` to access the URL for the sign up. This is on line 11.
4. On line 12 we make sure that we have a response that is `:success`, which corresponds to `200 OK`.
5. Then we use `assert_select` to make sure that the response includes expected HTML markup. Lines 14 - 16.
6. Then we send a `post` to the sign up path with the parameters that are supposed to be submitted with the sign up form. Line 19.
7. On line 20, we make sure that the response is redirect.
8. On line 22, we follow the redirect and we then land on the final response that it needs to be success (line 23).
9. Finally on line 26, we use `assert_select` again in order to make sure that the content is as expected.

Integration Test Interaction Methods

When you write integration tests, the full list of interaction helper methods that you can use is (including `get`, `post` and `follow_redirect!` that we have already seen):

1. `get`
2. `post`
3. `patch`
4. `put`
5. `delete`
6. `follow_redirect!`
7. `head`

Controller Tests

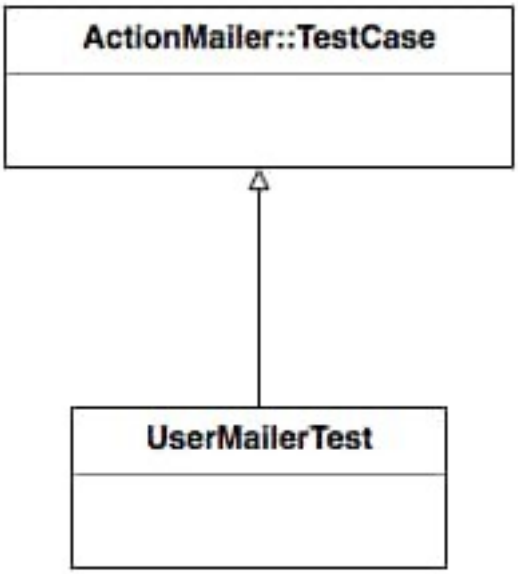
Controller tests used to be written inside `test/controllers` directory. Currently, this is deprecated. Rails does not favour controller tests. It favours integration tests instead. So, we will not expand on controller tests any further.

Note that controller tests used to be called functional tests too.

Testing Mailers

Let's see now how we can test our `UserMailer`. Note that in `test` environment the emails are not actually delivered, but they are kept inside a queue.

The tests for the `UserMailer` are going to be stored inside the folder `test/mailers` in the file named `user_mailer_test.rb`. Note that the `user_mailer_test.rb` should define a class that derives from `ActionMailer::TestCase`:



UserMailerTest Class Should Derive From ActionMailer::TestCase

Here is the code that we can use to test our `UserMailer`:

```
# File: test/mailers/user_mailer_test.rb
#
class UserMailerTest < ActionMailer::TestCase
  test 'ask for email confirmation' do
    user = users(:peter)
    email = UserMailer.ask_for_email_confirmation(user.id)

    assert_emails 1 do
      email.deliver
    end

    assert_equal ['support@techcareerbooster.com'], email.from
    assert_equal [user.email], email.to
    assert_equal 'TCB Blog: Please, Confirm Your Email Address', email.subject
    assert email.html_part.body.to_s.include?('Please, confirm your email address\
 by clicking on this link here')
  end
end
```

(the above code snippet online)

It is very straightforward:

1. On line 5, we use the fixture `:peter` to load a user.
2. On line 6, we send the `ask_for_email_confirmation` email to this user.
3. One lines 8 to 10 we send the email out and we make sure that the queue of emails has 1 element in.
4. Then we assert the parts of the email itself.
    1. On line 12, we check the `from` field.
    2. On line 13, we check the `to` field.
    3. On line 14, we check the `subject`.
    4. On line 15, we check the body of the email. Of course, for the body of the email you can include many more checks.

Note that Rails, while in `test` environment, it stores the emails generated during a test, inside the queue `ActionMailer::Base.deliveries`. This is an array and you can access it and ask questions about its content, if you want to assert that emails are being queued as expected.

So, the previous test could have been written as follows too:

```
1  # File: test/mailers/user_mailer_test.rb
2  #
3  class UserMailerTest < ActionMailer::TestCase
4    test 'ask for email confirmation' do
5      user = users(:peter)
6      email = UserMailer.ask_for_email_confirmation(user.id)
7
8      assert_difference "ActionMailer::Base.deliveries.size", 1 do
9        email.deliver
10     end
11
12     assert_equal ['support@techcareerbooster.com'], email.from
13     assert_equal [user.email], email.to
14     assert_equal 'TCB Blog: Please, Confirm Your Email Address', email.subject
15     assert email.html_part.body.to_s.include?('Please, confirm your email address\
16  by clicking on this link here')
17   end
18 end
```

(the above code snippet online)

The different line being line 8. Instead of `assert_emails 1 do`, we accessed the queue and we used the `assert_difference` assertion command.

An Integration Test That Email Is Sent Out

Besides the unit tests that we can write for an email action, we can also write an integration test that would make sure we send out the email at the correct point in the business flow.

Let's write an integration test to make sure that an email is sent out when a user signs up.

Open the file `test/integration/sign_up_test.rb` and add the new test `test 'email to confirm email address is sent out on sign up' do`:

```
  test 'email to confirm email address is sent out on sign up' do
    email = "#{SecureRandom.hex}@mailinator.com"
    test_password = '12345678'

    assert_difference 'ActionMailer::Base.deliveries.size', 1 do
      post sign_up_path, params: {user: {email: email, password: test_password, p\
assword_confirmation: test_password}}
    end

    email_sent = ActionMailer::Base.deliveries.last
    assert_equal [email], email_sent.to
  end
```

(the above code snippet online)

This is pretty straightforward. We wrap the `post sign_up_path` call into an `assert_difference` assertion. We also check that the last email in the queue has the correct properties.

Using RSpec

We have seen how we can use the test tools that come with Ruby on Rails. However, you may want to use, `RSpec` instead. How can you introduce `RSpec` into an existing Rails application?

Let's do it for our `blog` application.

This is the link to the gem that we should use to incorporate `RSpec` into our Rails project:

rspec-rails

Install Gem

Let's install the gem, by referencing it inside our `Gemfile`. In the `development` and `test` groups add the `gem 'rspec-rails'` statement, and then run `bundle`.

If, after `bundle` finishes, you do `bundle show | grep 'rspec'` you will see all the `RSpec` related gems that were installed in your blog bundle:

```
blog $ bundle show | grep 'rspec'
  * rspec-core (3.6.0)
  * rspec-expectations (3.6.0)
  * rspec-mocks (3.6.0)
  * rspec-rails (3.6.0)
  * rspec-support (3.6.0)
blog $
```

(the above code snippet online)

Initialize the `spec` Folder

Let's initialize the `RSpec` for our project. In order to do that, we will use a Rails generator that has been added to the list of Rails generators after having installed the `rspec` gems:

```
blog $ bin/rails generate rspec:install
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
blog $
```

(the above code snippet online)

As you can see, the `rspec:install` created the file `.rspec`, the folder `spec` and the files `spec/spec_helper.rb` and `spec/rails_helper.rb`.

Write Model Specs

Now that we have `RSpec` installed, we can write model specs using `RSpec`.

> **Important**: in order to be able to follow this section on `RSpec` you need to have studied the chapter *Introduction To RSpec* and *Useful RSpec Tools*, from the section *Test Automation, TDD and BDD*

Let's write the specs for the `User` model like we did earlier. Inside folder `spec/model` we create the file `user_spec.rb`:

```ruby
# File spec/models/user_spec.rb
#
require 'rails_helper'

describe User do
  it 'should not save user without email' do
    user = User.new(password: '12345678')

    expect(user.save).to eq(false)
    expect(user.errors[:email]).to include("can't be blank")
  end

  it 'generates email confirmation token' do
    # setup
    user = User.new(email: 'foo@bar.com', password: '12345678')

    expect(user.email_confirmation_token).to be_blank

    # fire
    user.save!

    # check results
    expect(user.email_confirmation_token).not_to be_blank
  end
end
```

(the above code snippet online)

The above is the exact equivalent to the `test/models/user_test.rb`, but it is an `RSpec` specification file. Since you already know `RSpec`, working out this code is trivial.

Write Integration Specs

We can use `RSpec` to write integration tests (or *request* specs as `RSpec` might call them).

Let's write the `test/integration/sign_up_test.rb` but in its `RSpec` version.

Go to `spec` and create the folder `integration`. Then create the file `sign_up_spec.rb` as follows:

```ruby
# File: spec/integration/sign_up_spec.rb
#
require 'rails_helper'

describe 'Sign Up Test' do
  it 'sign up happy path' do
    email = "#{SecureRandom.hex}@mailinator.com"
    test_password = '12345678'

    # 1. send a GET request to the sign up path
    get sign_up_path
    expect(response).to be_success

    assert_select 'input[name="user[email]"]'
    assert_select 'input[name="user[password]"]'
    assert_select 'input[name="user[password_confirmation]"]'

    # 2. send a POST request to the sign up path, and follow the redirect
    post sign_up_path, params: {user: {email: email, password: test_password, pas\
sword_confirmation: test_password}}
    expect(response).to be_redirect

    follow_redirect!
    expect(response).to be_success

    # 3. check now what is inside the HTML content
    assert_select '.alert.alert-success', /You have successfully signed up! Pleas\
e, confirm your email address\. We just sent you out an email with a link to do s\
o!/
  end

  it 'email to confirm email address is sent out on sign up' do
    email = "#{SecureRandom.hex}@mailinator.com"
    test_password = '12345678'

    expect do
      post sign_up_path, params: {user: {email: email, password: test_password, p\
assword_confirmation: test_password}}
    end.to change {ActionMailer::Base.deliveries.size}.by(1)

    email_sent = ActionMailer::Base.deliveries.last
    expect(email_sent.to).to eq([email])
  end
end
```

(the above code snippet online)

The above is the content of the file `spec/integration/sign_up_spec.rb` and it is completely equivalent to the test functionality offered by `test/integration/sign_up_spec.rb`. Again, it is easy to understand if you know how `RSpec` works.

Note also that the `assert_select` methods that are used here are methods that are coming from `ActionDispatch::Assertions` which are mixed in to `RSpec` integration example groups.

Write Feature Specs

The `RSpec` Feature Specs are the equivalent of the system tests that we saw earlier. The default is that they are written inside the `spec/features` folder and not inside the `spec/system` folder. Although, you can write them in `spec/system` folder and tag them with the `type: :feature` so that `RSpec` knows that they are feature specs.

Let's create the equivalent of the `test/system/sign_up_test.rb`. Create the folder `spec/system` and then create the file `sign_up_spec.rb` with the following content:

```ruby
# File: spec/system/sign_up_spec.rb
#
require 'rails_helper'

describe 'Sign Up Test', type: :feature do
  it 'sign up happy path' do
    email = "#{SecureRandom.hex}@mailinator.com"
    test_password = '12345678'

    visit root_url

    click_link 'Sign Up'

    fill_in 'Email', with: email
    fill_in 'Password', with: test_password
    fill_in 'Password confirmation', with: test_password

    click_button 'Sign Up'

    assert_selector '.alert.alert-success', text: 'You have successfully signed u\
p! Please, confirm your email address. We just sent you out an email with a link \
to do so!'
  end
end
```

(the above code snippet online)

This is the same like the equivalent `test/system/sign_up_test.rb` file.

RSpec and Fixtures

RSpec can read the fixtures that you have set up, if you want. But you need to make sure of the following:

1. The fixtures path is correctly configured for RSpec. Look in the file `rails_helper.rb`. There, you will see a line `config.fixture_path = "#{::Rails.root}/spec/fixtures"` which points to the folder `spec/fixtures` for the fixtures. So, you have to have your fixtures in this folder or, otherwise, you will have to update this value accordingly.
2. You will need to call `fixtures <fixtures collection>` as the first command of the describe block in which you will be using the fixture commands.E.g.

```
 describe User do
   fixtures :users
   ...
   # then you are allowed to call methods like `peter = users(:peter)` inside you\
r `it` blocks.
   ...
 end
```

Using Cucumber

Let's now see how we can introduce Cucumber into our Rails test suite. We have learnt about Cucumber in Section "Test Automation, TDD and BDD", Chapters "Introduction to Cucumber" and "Advanced Cucumber". So, we will not delve into too many details here.

The easiest way for you to integrate Cucumber to your Rails project is to use cucumber-rails gem.

Installing Gem

Open your `Gemfile` and inside the `:test` group add the following lines:

```
gem 'cucumber-rails', :require => false
# database_cleaner is not required, but highly recommended
gem 'database_cleaner'
```

(the above code snippet online)

> *Note:* we also bring in the gem `database_cleaner` which is highly recommended when you work with Rails applications that use a database.

Then, run `bundle` to install the required gems.

After the end of this command, if you run `bundle show | grep 'cucumber'`, you will see something like the following:

```
blog $ bundle show | grep 'cucumber'
  * cucumber (2.4.0)
  * cucumber-core (1.5.0)
  * cucumber-rails (1.5.0)
  * cucumber-wire (0.0.1)
blog $
```

(the above code snippet online)

These are the `cucumber` related gems that have just been installed.

Initializing Cucumber

Then you need to initialize Cucumber. In order to do that, you need to invoke the corresponding Rails generator.

```
blog $ bin/rails generate cucumber:install
      create  config/cucumber.yml
      create  script/cucumber
       chmod  script/cucumber
      create  features/step_definitions
      create  features/step_definitions/.gitkeep
      create  features/support
      create  features/support/env.rb
       exist  lib/tasks
      create  lib/tasks/cucumber.rake
        gsub  config/database.yml
        gsub  config/database.yml
       force  config/database.yml
blog $
```

(the above code snippet online)

which does many things. But the most important for you at the moment is the `features` folder in which you are going to write your Feature files and the `features/step_definitions` folder in which you are going to write the implementation of the steps.

Model Features

With Cucumber we can specify the requirements of our Web application, but also, we can specify the requirements of the domain model. This means that we can use Cucumber/Gherkin to write the requirements of the Models of our Rails application.

Let's do that. We will create the folder `models` inside `features` folder. Then we will create the folder `users`, inside the `models` folder. This `user` folder will be holding Feature files that would describe the requirements of the `User` entity.

Let's create the file `user_validation_rules.feature` that will include the requirements that have to do with validation of a `User` entity:

```
# File: features/models/user/user_validation_rules.feature
#
Feature: User validation rules

  Scenario: User that does not have an email
    Given a user without email
    Then user is not valid
    And has errors on email
```

(the above code snippet online)

Let's run this one, but just calling `cucumber` on the command line:

```
blog $ cucumber
Using the default profile...
# File: features/models/user/user_validation_rules.feature
#
Feature: User validation rules

  Scenario: User that does not have an email, is not valid and cannot be saved # \
features/models/user/user_validation_rules.feature:5
    Given a user without email                                                  # \
features/models/user/user_validation_rules.feature:6
      Undefined step: "a user without email" (Cucumber::Undefined)
      features/models/user/user_validation_rules.feature:6:in `Given a user witho\
ut email'
    Then user is not valid                                                      # \
features/models/user/user_validation_rules.feature:7
      Undefined step: "user is not valid" (Cucumber::Undefined)
      features/models/user/user_validation_rules.feature:7:in `Then user is not v\
alid'
    And has errors on email                                                     # \
features/models/user/user_validation_rules.feature:8
      Undefined step: "has errors on email" (Cucumber::Undefined)
      features/models/user/user_validation_rules.feature:8:in `And has errors on \
email'

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.187s

You can implement step definitions for undefined steps with these snippets:

Given(/^a user without email$/) do
  pending # Write code here that turns the phrase above into concrete actions
end

```

```
Then(/^user is not valid$/) do
  pending # Write code here that turns the phrase above into concrete actions
end

Then(/^has errors on email$/) do
  pending # Write code here that turns the phrase above into concrete actions
end

blog $
```

(the above code snippet online)

You can see that `cucumber` tells you that the steps are not defined.

Let's do the step definitions. Create the file `features/step_definitions/user_steps.rb` as follows:

```
# File: features/step_definitions/user_steps.rb
#
Given(/^a user without email$/) do
  test_password = '12345678'
  @user = User.new(password: test_password, password_confirmation: test_password)
end

Then(/^user is not valid$/) do
  expect(@user).to be_invalid
end

Then(/^has errors on email$/) do
  expect(@user.errors[:email]).to include("can't be blank")
end
```

(the above code snippet online)

This is pretty much straightforward. Note that for the expectations we use RSpec commands here.

Integration Tests

We don't use `Cucumber` to write integration tests. However, we use `Cucumber` to write System requirements/tests/specs.

System Requirements

Let's now proceed with writing an end-to-end scenario with `Cucumber`. It is going to be the Sign Up feature.

Inside the `features/system` create the file `sign_up.feature` and put the following content inside:

```
# File: features/system/sign_up.feature
#
Feature: Sign Up

  Scenario: Sign Up Happy Path
    Given I visit the Web application
    And I decide to sign up
    When I fill in my email and password
    And I submit my details
    Then I can see a confirmation message that I have signed up successfully
```

(the above code snippet online)

This is the happy path for Sign Up feature. If you run `cucumber`, you will get the steps undefined, of course. Let's copy paste the suggested step definitions inside the file `features/step_definitions/sign_up_steps.rb`:

```
# File: features/step_definitions/sign_up_steps.rb
#
Given(/^I visit the Web application$/) do
  visit root_url
end

Given(/^I decide to sign up$/) do
  click_link 'Sign Up'
end

When(/^I fill in my email and password$/) do
  fill_in 'Email', with: "#{SecureRandom.hex}@gmail.com"
  fill_in 'Password', with: '12345678'
  fill_in 'Password confirmation', with: '12345678'
end

When(/^I submit my details$/) do
  click_button 'Sign Up'
end

Then(/^I can see a confirmation message that I have signed up successfully$/) do
  assert_selector '.alert.alert-success', text: 'You have successfully signed up!\
 Please, confirm your email address. We just sent you out an email with a link to\
 do so!'
end
```

(the above code snippet online)

Compare the above implementation with the one that we did in file `test/system/sign_up_test.rb`. You will see that they are the same. Only that we now use Gherkin, which, for some people, is considered much more easy for non-tech people to understand.

Before we close, let me tell you that Capybara API includes RSpec Matchers that you can use for Querying whether page contains specific content. So, instead of

```
assert_selector '.alert.alert-success', text: 'You have successfully signed up! P\
lease, confirm your email address. We just sent you out an email with a link to d\
o so!'
```

(the above code snippet online)

which uses the `assert_selector` from `Capybara::Node::Matchers`, you can use the following:

```
expect(page).to have_selector('.alert.alert-success', text: 'You have successfull\
y signed up! Please, confirm your email address. We just sent you out an email wi\
th a link to do so!')
```

(the above code snippet online)

which uses the corresponding RSpec Matcher from `Capybara::RSpecMatchers`.

> *Information:* `Capybara::Node::Matchers` and `Capybara::RSpecMatchers` are being included in Rails system and integrations tests.

Factory Girl

Before we finish this introduction to Rails testing, I would like to tell you about Factories and in particular about the Factory Girl. Factory Girl offers an API/DSL to allow to quickly construct instance of objects at the beginning of each of your tests.

Note that many confuse Fixtures to Factories provided by Factory Girl, but these are different concepts. Fixtures are not supposed to be there to serve a particular test, where as factory instances are. In other words, we use fixtures to set up the database with all the necessary data to run all the tests and not any specific test in particular. They are data that need to be there for all tests. On the other hand, we use factory instances to prepare the data for a specific test.

Also, some use Factories to set up global database data in a before suite hook. In that case, they basically do what the fixtures do with the `yml` files.

Let's see how we can use `factory_girl` to eliminate some code duplication from our tests.

Install `factory_girl_rails`

In a Rails application, if we want to use `factory_girl`, we do it via the gem factory_girl_rails.

Let's add the following line inside our `Gemfile` file, inside the `:development, :test` groups:

```
gem 'factory_girl_rails'
```

(the above code snippet online)

Then run `bundle`. This will install all the necessary gems for you to use `factory_girl`:

```
blog $ bundle show | grep 'factory'
  * factory_girl (4.8.0)
  * factory_girl_rails (4.8.0)
blog $
```

(the above code snippet online)

Use a Factory in `User` Model Test

Let's open the file: `test/models/user_test.rb`. You can see that we have invocation to create `User` instances at the beginning of both files. What if we just had this:

```
1  # File: test/models/user_test.rb
2  #
3  require 'test_helper'
4
5  class UserTest < ActiveSupport::TestCase
6    test 'should not save user without email' do
7      user = FactoryGirl.build(:user, email: nil)
8
9      assert_not user.save
10     assert_includes user.errors[:email], "can't be blank"
11   end
12
13   test 'generates email confirmation token' do
14     # setup
15     user = FactoryGirl.build(:user)
16
17     assert user.email_confirmation_token.blank?
18
19     # fire
20     user.save!
21
22     # check results
23     assert_not user.email_confirmation_token.blank?
24   end
25 end
```

(the above code snippet online)

Please, have a look on lines 7 and 15. They both use the `FactoryGirl.build` API from Factory Girl to create a user. The method `FactoryGirl.build` takes as first input argument the name of the factory to be used. Then it may take attributes of the instance to override. For example, the `FactoryGirl.build(:user, email: nil)` will create a instance of a `User` without an email. Whereas the `FactoryGirl.build(:user)` will create an instance of `User` with all attributes as defined in the factory.

The factories are defined inside the `test/factories` folder. So, let's create the file `test/factories/users.rb` and put the following content inside:

```
1  # File: test/factories/users.rb
2  #
3  FactoryGirl.define do
4    factory :user do
5      email { "#{SecureRandom.hex}@mailinator.com" }
6      password '12345678'
7    end
8  end
```

(the above code snippet online)

The above defines the factory with name `:user` between lines 4 to 7. The definition comes inside a `FactoryGirl.define do...end` block. Inside this block you could have defined many other factories.

Now, let's run our test suite again:

```
1  blog $ bin/rails test
2  Running via Spring preloader in process 51457
3  Run options: --seed 32286
4
5  # Running:
6
7  .....
8
9  Finished in 2.900929s, 1.7236 runs/s, 6.5496 assertions/s.
10 5 runs, 19 assertions, 0 failures, 0 errors, 0 skips
```

(the above code snippet online)

You will see all the tests succeeed.

Because typing `FactoryGirl.` before each method call to `FactoryGirl` is a lot of repetition, you can mix in the module `FactoryGirl::Syntax::Methods` inside your `ActiveSupport::TestCase`. Then, all the `FactoryGirl` methods will be available without you having to prefix their calls with `FactoryGirl.`.

```
1  require File.expand_path('../../config/environment', __FILE__)
2  require 'rails/test_help'
3
4  class ActiveSupport::TestCase
5    # Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
6    fixtures :all
7
8    # Add more helper methods to be used by all tests here...
9    include FactoryGirl::Syntax::Methods
10 end
```

(the above code snippet online)

Do you see line 9? This is how we mix in the `FactoryGirl` methods. Then, you can turn `FactoryGirl.build` method calls to just `build` method calls:

```
1  # File: test/models/user_test.rb
2  #
3  require 'test_helper'
4
5  class UserTest < ActiveSupport::TestCase
6    test 'should not save user without email' do
7      user = build(:user, email: nil)
8
9      assert_not user.save
10     assert_includes user.errors[:email], "can't be blank"
11   end
12
13   test 'generates email confirmation token' do
14     # setup
15     user = build(:user)
16
17     assert user.email_confirmation_token.blank?
18
19     # fire
20     user.save!
21
22     # check results
23     assert_not user.email_confirmation_token.blank?
24   end
25 end
```

(the above code snippet online)

You can now see that lines 7 and 15 use `build` instead of `FactoryGirl.build`. If you now run your tests again, they will succeed as before.

Note that `build` create an instance in memory, whereas `create` creates an instance both in memory and persistently in the database. The document here is an excellent resource on Factory Girl.

Factory Girl And RSpec

If you want to use Factory Girl in your RSpec specifications, you will need to add `config.include FactoryGirl::Syntax::Methods` inside the `RSpec` configuration in your `spec/rails_helper.rb` file.

Factory Girl And Cucumber

If you want to use Factory Girl API in your Cucumber step definitions without prefixing with `FactoryGirl.`, you need to add at the bottom of your `features/support/env.rb` file the statement `World(FactoryGirl::Syntax::Methods)`.

> Note that `World` is a method even if its name starts with an uppercase letter

Closing Note

We have seen the main tools used in Rails to test our Web application. The domain model using model tests, the integration tests to see how the HTTP requests hit and are being served by our REST server. The system tests that may be used to test how the user interacts with our WEB application. As we proceed and learn more about Rails we will learn more things about testing too.

Important Note that depending on the strategy of the development and/or testing team that you work in, you will most probably not required to write all the categories of the tests that you learned in this chapter, neither, necessarily, with all the tools like `RSpec`, `Cucumber`, `Capybara`, `FactoryGirl` and fixtures. Also, the more tests you write the better you become and you decide what kind of tests you want to write. There is no right or wrong.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

Make sure that you have all the things that the chapters teaches you incorporated into your `blog` application. Since the chapter content didn't cover for all tests that one could write for the `blog` application, make sure that you do the following as well:

1. Write the following tests for the `User` model:

```
test 'cannot have an email that is already used' do

end

test 'cannot have an email that is already used even with different case' do

end

test 'user has email confirmed if there is no email confirmation token' do

end

test 'user does not have email confirmed if there is email confirmation token' do

end

test 'user can write an Article if it is their own Article' do

end

test 'user cannot write an Article if it is not their own Article' do

end
```

2. `RSpec`: The above tests need to be amended inside the `spec/models/user_spec.rb` file too.
3. `Cucumber`: The above tests need to be amended inside the file `features/models/user/user_-validation_rules.feature`.
4. Write tests for the model 'SignIn'. Cover for email presence, password presence and the authenticate functionality.
5. You need to write 'SignIn' tests in `RSpec` too.
6. You need to write `SignIn` tests in `Cucumber` too.
7. You need to write `Article` model tests. Cover for user presence, title presence and text presence.
8. You need to write `Article` model tests in `Rspec` too.
9. You need to write `Article` model tests in `Cucumber` too.
10. Write system tests for the feature of creating a new Article. Tests should be written inside `features/system/articles_new.feature` file using `Cucumber`. We expect you to write two scenarios:
    1. User that is not signed in cannot create an Article
    2. User that is signed in can create an Article
11. Write system tests for the feature of editing an existing Article. Tests should be written inside `features/system/articles_edit.feature` file using `Cucumber`. We expect you to write the following scenarios:
    1. User that is not signed in cannot edit an Article
    2. User that is signed in cannot edit the Article of another User
    3. User can edit their own Article

Hints:

1. When you want to see what errors an attribute has, use the `puts` command to print the `.errors` hash. E.g. `puts user.errors[:email].inspect` will print the errors `user` instance might have on the `email` attribute.
2. Learn how to create Factory Girl associations. You will find it useful to create the `:article` factory. Factory Girl associations are explained here (look for the section `Associations`).
3. `page.current_path` returns the current path your system test is on.
4. `Article.last` returns the last `Article` created.

7 - More About Models

Summary

More About Models - Summary

The Models are the heart of your Web application, because they model your business domain in the object / Ruby world. This is an extensive chapter on Rails Models. You will learn all the necessary details that will make you work with Models more efficiently and effectively.

Learning Goals

1. Learn about Active Record pattern.
2. Learn about Active Record conventions.
3. Learn about the format of class names for Active Record models and their equivalent table name.
4. Learn how Rails can correctly convert singular to plural.
5. Learn how to convert a CamelCase string to underscore format.
6. Learn how primary keys and foreign keys are conventionally named.
7. Learn how to override Rails/Active Record naming conventions.
8. Learn the Active Record statements that allow you to create, retrieve, update and delete records from the database.
9. Learn about the various powerful finding methods of Active Record.
10. Learn about the Active Record Relations.

11. Learn about callbacks.
12. Learn about schema migrations.
13. Learn about the schema migrations timestamps and how Rails keeps track of which migrations have been executed.
14. Learn how to take back/destroy migrations.
15. Learn about the `db/schema.rb` file.
16. Learn about creating joined tables.
17. Learn how to create association tables using `create_join_table`.
18. Learn how to execute custom SQL in your migrations.
19. Learn about the Validations and which methods trigger their execution.
20. Learn how to check whether an object is valid or invalid.
21. Learn how to get the validation errors attached to an object.
22. Learn how to triggers validations, selectively, either on creation of a record or on update of it.
23. Learn more details about specific validations, like the *acceptance* validation.
24. Learn how to customize the validation error messages.
25. Learn how you can write custom validators.
26. Learn about the common options for all the validators.
27. Learn about the conditional validations.
28. Learn about the Active Record callbacks.
29. Learn about the order in which callbacks are called.
30. Learn about the special callback `after_initialize`.
31. Learn about the special callback `after_find`.
32. Learn about the special callback `after_touch`.
33. Learn how to skip Active Record callbacks.
34. Learn how you can halt the execution of a create or an update.
35. Learn about the relational callbacks.
36. Learn about the conditional callbacks.
37. Learn about the transactional callbacks.
38. Learn more about Active Record associations.
39. Learn about which methods are implicitly defined when declaring associations.
40. Learn about the `has_many` and `through` association.
41. Learn more about the Active Record query interface.
42. Learn how to fetch records in batches.
43. Learn to use conditions when fetching records.
44. Learn how to ask your database to return ordered results.
45. Learn how to select specific columns to be returned in the result set.
46. Learn how you can use pagination to bring records from your database.
47. Learn how to incorporate *group by*.
48. Learn how to incorporate *having*.
49. Learn about the available locking mechanisms.
50. Learn how you can join columns from different tables.
51. Learn about eager loading and how you can avoid the `N + 1` queries problem.
52. Learn about scopes.
53. Learn about dynamic finders.

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

54. Learn about enums.
55. Learn about the *find or create by* methods.
56. Learn how to use raw SQL to fetch records from the database.
57. Learn about `pluck` and how it does differ from `select`.
58. Learn about the `ids` method.
59. Learn how you can check whether a record exists, without actually fetching any data from the database.
60. Learn about the Calculation functions.

Introduction

We start our study of Models in Ruby on Rails projects with the Models that are Active Records, i.e. they derive from `ActiveRecord::Base`. *ActiveRecord::Base* is the implementation of the *Active Record* pattern, as described by Martin Fowler here. It basically says that an instance of an Active Record:

1. Has state that needs to be persisted into a database.
2. It has methods that can persist the state into the database.
3. It has business/domain logic that operates on the state.

Active Record is a tool that allows us to map objects to tables into a relational database. These tools are called ORM tools, i.e. Object-Relation Mapping tools. With Active Record we write Ruby and not complex SQL to retrieve and update data into our database.

Active Record Quick Overview

Let's start with a quick overview. After the overview, we will elaborate the areas that need better study.

Active Record Conventions

Active Record, like the rest of Ruby on Rails, uses a lot of conventions (over configuration) in order to make you write less code and less configuration files.

For example, when you write:

```
class Article < ApplicationRecord
end
```

(the above code snippet online)

i.e. when you make the class `Article` derive from `ActiveRecord::Base` (note that `ApplicationRecord` derives from `ActiveRecord::Base`), then Rails knows that this is going to be a class that would map to the table that has the same name but in plural form and lower case letters. In other words, `Article` class would map to `articles` table.

Also, if you have the model:

```
class OrderItem < ApplicationRecord
end
```

(the above code snippet online)

then, Rails, knows that this class should map to the table `order_items`.

The model class names follow the *Camel Case* format, whereas the corresponding table name follows the *snake* format. In *Camel Case* format, the names start with an uppercase letter for each new word, whereas in *snake* format the words are separated using underscores.

In the following table, you can see some examples of proper model class names and their corresponding table name:

Model/Class Name	Table Name
`Customer`	`customers`
`OrderItem`	`order_items`
`Person`	`people`

See how Rails is clever enough to convert `Person` to `people` when looking for the table name. You can see how Rails pluralizes words, by exercising the method `#pluralize` on strings. Try that with a Rails console on your `blog` project:

```
blog $ bin/rails console
Loading development environment (Rails 5.1.1)
2.3.3 :001 > 'person'.pluralize
 => "people"
2.3.3 :002 > 'basis'.pluralize
 => "bases"
2.3.3 :003 > 'hand'.pluralize
 => "hands"
2.3.3 :004 > 'appendix'.pluralize
 => "appendixes"
2.3.3 :006 > 'child'.pluralize
 => "children"
2.3.3 :007 >
```

(the above code snippet online)

Another method that you can try on strings and that it is related to our discussion is the method `#underscore`. This converts a `CamelCase` to its snake equivalent:

```
2.3.3 :007 > "CamelCase".underscore
 => "camel_case"
2.3.3 :008 >
```

(the above code snippet online)

Besides the table name conventions, Active Record uses some conventions for the column names too. For example, all the primary keys will be stored in a column named `id`. Also, the foreign

keys, they will have the name that denotes the destination table in singular form suffixed by an `_id`. Look for example the `Article` model with table `articles`. It has the column `user_id` as a foreign key column pointing to `users` table `id` primary key of that table. Having done that, when we write `belongs_to :user`, `ActiveRecord::Base` knows that the underlying persisting information is stored in the column `user_id`.

Creating Active Record Models

We have seen how we can create Active Record models. It was very easy. Wasn't it? We have created the `User` and the `Article` models.

Overriding The Naming Conventions

But how can you create a model for a legacy application that does not follow the conventions that Rails/Active Record relies on?

For example, let's assume that you have a table named `bkorder` in which you would like to save `Order` data. In other words, your model in your object world is `Order`, but your table, has been named `bkorder`. You need to tell Rails that the corresponding table is not `orders`, but it is `bkorder`. You can do that like in the following code:

```
class Order < ApplicationRecord
  self.table_name = 'bkorder'
end
```

(the above code snippet online)

Or, you may want to use a different primary key name for the primary key column. Let's suppose that the table `bkorder` has primary key column named `bkorder_identifier`. You can speficy the primary key as follows:

```
class Order < ApplicationRecord
  self.table_name = 'bkorder'
  self.primary_key = 'bkorder_identifier'
end
```

(the above code snippet online)

Creating, Retrieving, Updating and Deleting (CRUD) Records.

Creating

There are many ways you can use to create a new record into the database. Let's take for example, the `User` model from our `blog` application.

Method 1

```
user = User.create(email: 'foo@mailinator.com', password: '12345678')
```

(the above code snippet online)

The `User.create` will take as input a `Hash` with the attributes and their values. It will create the record in the database and will return an instance of `User` class.

If the record cannot be created for any reason, this method will return `false`. There is also the method `User.create!` that you can use when you want an exception to be raised in case of a failure. I usually use this when writing tests so that I can quickly see the failing point in my test.

Method 2

```
user = User.new
user.email = 'foo@mailinator.com'
user.password = '12345678'
user.save
```

(the above code snippet online)

Here, we use the method `save` instead of `create`. When using `save` on the instance of a `User`, if the instance corresponds to a new record, then the `save` will actually do a `create`.

The `user.save` will return `false` if something goes wrong. And again, if you want, you can use the method `#save!` instead. In that case, it will raise an exception if the save fails. Useful when writing tests.

Retrieving / Reading a Record From Database

Active Record is very powerful in terms of retrieving data from the database, executing SQL queries behind the scenes, while we write Ruby object code. Here are some examples:

Retrieve All Records

```
articles = Article.all
```

(the above code snippet online)

Calling `.all` on the model class returns an `ActiveRecord::Relation` instance that will allow you to go through all the records of this model in the database.

Count Records

```
Article.count
```

(the above code snippet online)

Returns the number of records in the database.

First Record

```
Article.first
```

(the above code snippet online)

This returns the first record in the database after ordering the records on the primary key. In other words, the above Active Record statement executes a `select * from articles order by id limit 1` in order to bring the first record as a result.

Last Record

```
Article.last
```

(the above code snippet online)

Returns the last record when ordering by the primary key.

Finding By An Attribute

```
Article.find_by(title: 'History')
```

(the above code snippet online)

It will return any Article that has title with value `History`. Note that even if two or more Articles have the value `History` in their title, only one will be returned. If you want to get all the Articles that have the value `History` in their title, you should use the `where` method that we explain below.

Be aware that depending on the RDBMS vendor that you use and how it has been set up, `.find_by` may not find the records that you would expect, because it does exact search. But if your RDBMS translates the exact search to a case insensitive search your search is going to be more relaxed.

In other words, if your database has two Articles, one with title `History` and another one with title `history` the `Article.find_by(title: 'History')` will only bring the first one, if your database is PostgreSQL, but it might bring any of the two with the default MySQL set up. Of course, the issue is more obvious when you try the `.where` method that returns an array of Articles.

Retrieving With Conditions / Where Clause

```
Article.where(title: 'History')
```

(the above code snippet online)

It will retrieve the Articles that have title `History`. The result is an `ActiveRecord::Relation` which you can use to iterate over the matching Articles:

```
history_articles = Article.where(title: 'History')
history_articles.each do |article|
  puts article.text
end
```

(the above code snippet online)

In the above example, we iterate, using `each`, over all the Articles that have title `History`.

Later on, we will elaborate on more complex retrieving/query examples.

Updating a Record

Updating a record using Active Record is super easy.

```
article.title = 'About Geography'
article.save
```

(the above code snippet online)

You just set the new values on the attributes of the instance and then you call `#save`. You can also call `#save!` if you want to get an exception if something goes wrong.

Also, you can update many attributes with one Ruby call. In order to do that, you need to use method `#update`.

```
article.update(title: 'About Geography', text: 'We will talk about Geography...')
```

(the above code snippet online)

Deleting A Record

Deleting a record is equally simple:

```
article = Article.find_by(title: 'History')
article.destroy
```

(the above code snippet online)

The call to `#destroy` will execute a `DELETE` statement in the database.

You can also call `.destroy_all` to delete all records from the database. But, be very careful with that. It will really delete all records and you will not be able to recover unless you have a recent backup of your database.

```
Article.destroy_all
```

(the above code snippet online)

CallBacks

Another very important aspect of models are the model lifecycle callbacks. We have seen such a callback in the `User` model of the `blog` application:

```
before_create :generate_email_confirmation_token
```

(the above code snippet online)

Callbacks allow us to hook code at specific points in the lifecycle of a Model instance. I will give you more details about them a little bit later on.

End of Overview

That was a quick overview of the most important things for Active Record models. We continue with more details explanation on some of them below.

Schema Migrations

We have already seen what schema migrations are, while working with our `blog` application. We have learned:

1. How to create a table with the `create_table` command
2. How to define the columns of the table.
3. How to defined timestamps columns, `created_at` and `updated_at`.
4. How to write a schema migration to add a column to a table.
5. How to create an index on a table.
6. How to add foreign keys.
7. How we can write the rollback part of a forward schema migration.

Let's now see some more details about schema migrations.

Timestamp

When we create a schema migration with the corresponding rails generator the file created has a unique timestamp.

Let's do an example in our `blog` application:

```
blog $ bin/rails generate migration create_reviews
      invoke  active_record
      create    db/migrate/20170701174349_create_reviews.rb
blog $
```

(the above code snippet online)

The timestamp is the prefix to the file generated. On my end, it was `20170701174349`. `CCYYMMDDHHMMSS` is the format of the timestamp and it is a UTC time.

Rails Keeps Track of Migrations Executed

This makes each migration unique. And Rails keeps track of which migrations have been executed and which not using a table called `schema_migrations` inside our database.

Let's confirm this with a `psql` session for our `blog_development` database:

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \connect blog_development                                  \
                                                                                    \
                                                                                    \
                  You are now connected to database "blog_development" as user "p\
anayotismatsinopoulos".
blog_development=# select * from schema_migrations;                                 \
                                                                                    \
                                                                                    \
                     version
----------------
 20170519062431
 20170604052426
 20170610052034
 20170617084556
 20170618060653
(5 rows)

blog_development=#
```

(the above code snippet online)

As you can see, the table has the timestamps of the previous 5 migrations that we have already executed. But, it does not have the timestamp for the latest migration that we have just generated. This means that Rails knows which migrations to run, if we execute `bin/rake db:migrate`. It will only run the migration `20170701174349`, i.e. the migration that we have just generated.

Before we migrate, let's see the content of the file of the pending schema migration:

```
class CreateReviews < ActiveRecord::Migration[5.1]
  def change
    create_table :reviews do |t|
    end
  end
end
```

(the above code snippet online)

Rails, having analyzed the `create_reviews` name of the migration has guessed the content and generated a migration that creates a table, named `reviews`. In order to continue this exercise, we are happy to keep this migration as is.

Let's now run the migration from the operating system shell prompt at the root folder of our `blog` application.

```
blog $ bin/rake db:migrate
== 20170701174349 CreateReviews: migrating ====================================
-- create_table(:reviews)
   -> 0.0441s
== 20170701174349 CreateReviews: migrated (0.0443s) ===========================

blog $
```

(the above code snippet online)

The migration has been executed successfully. Can we confirm the content of the `schema_migrations` table? Let's do that with the help of `psql`.

```
blog $ psql
psql (9.4.4)
Type "help" for help.

panayotismatsinopoulos=# \connect blog_development
You are now connected to database "blog_development" as user "panayotismatsinopou\
los".
blog_development=# select * from schema_migrations;
    version
----------------
 20170519062431
 20170604052426
 20170610052034
 20170617084556
 20170618060653
 20170701174349
(6 rows)

blog_development=#
```

(the above code snippet online)

Nice! A sixth row has been added, that of the latest migration that we have executed.

Now, let's go back to our operating system shell prompt and execute a rollback, so that we take back the latest migration that we have just executed.

```
blog $ bin/rake db:rollback
== 20170701174349 CreateReviews: reverting ===================================
-- drop_table(:reviews)
   -> 0.0061s
== 20170701174349 CreateReviews: reverted (0.0098s) ==========================

blog $
```

(the above code snippet online)

You can see the `drop_table(:reviews)` command has been executed.

What do we expect to have happened in the `schema_migrations` table? We expect it to have 1 row less now:

```
blog $ psql -d blog_development -c "select * from schema_migrations"
    version
----------------
 20170519062431
 20170604052426
 20170610052034
 20170617084556
 20170618060653
(5 rows)

blog $
```

(the above code snippet online)

> *Hint*: The `psql -d blog_development -c "select * from schema_migrations"` is a way to execute a SQL statement without actually starting an interactive session with `psql`.

And the output confirms our assumption. The migration `20170701174349` has been removed.

Cool! So, we have now learned how Rails keeps track of the migrations. Bare in mind that what you do in your development database has nothing to do with what happens in your production database or you test database. In other words, running a schema migration while in development environment will not affect your production database. You will have to run the migration again in your production environment in order for the migration to affect your production database.

A File With Current Schema Status

Note that every time you successfully run a migration, or a rollback, the file `db/schema.rb` is generated. Let's have a look at the `db/schema.rb` file of our `blog` application as it is supposed to be after having executed 5 of the 6 migrations:

```
1  # This file is auto-generated from the current state of the database. Instead
2  # of editing this file, please use the migrations feature of Active Record to
3  # incrementally modify your database, and then regenerate this schema definition.
4  #
5  # Note that this schema.rb definition is the authoritative source for your
6  # database schema. If you need to create the application database on another
7  # system, you should be using db:schema:load, not running all the migrations
8  # from scratch. The latter is a flawed and unsustainable approach (the more migra\
9  tions
10 # you'll amass, the slower it'll run and the greater likelihood for issues).
11 #
12 # It's strongly recommended that you check this file into your version control sy\
13 stem.
14
15 ActiveRecord::Schema.define(version: 20170618060653) do
16
17   # These are extensions that must be enabled in order to support this database
18   enable_extension "plpgsql"
19
20   create_table "articles", force: :cascade do |t|
21     t.string "title", null: false
22     t.string "text", null: false
23     t.datetime "created_at", null: false
24     t.datetime "updated_at", null: false
25     t.bigint "user_id"
26     t.index ["user_id"], name: "articles_user_id_idx"
27   end
28
29   create_table "users", force: :cascade do |t|
30     t.string "email", null: false
31     t.string "password_digest", null: false
32     t.datetime "created_at", null: false
33     t.datetime "updated_at", null: false
34     t.string "email_confirmation_token"
35     t.index ["email"], name: "users_email_uidx", unique: true
36   end
37
38   add_foreign_key "articles", "users", name: "articles_user_id_users_id_fk"
39 end
```

(the above code snippet online)

The main things that you need to know about this file are the following:

1. The latest migration timestamp can be found on line 13.
2. Then you can see all the `ActiveRecord::Migration` commands that would bring your database to the latest state.

As you can read from the comments in the file, this file is the source of truth with regards to your database schema. So, they say that if you want to recreate the database in another instance, you should use the `bin/rake db:schema:load` command, rather than just running `bin/rake db:migrate`.

Note that `bin/rake db:schema:load` is executed whenever you run your tests in order for the test runner to prepare the database for the tests to run.

Destroy Generated Migration

The `bin/rails generate migration` is used to generate a migration file. There is an opposite command too. The `bin/rails destroy migration` which is used to take back the result of the `generate` action.

Let's use it to take back the `CCYYMMDDHHMMSS_create_reviews.rb` generated file:

```
blog $ bin/rails destroy migration create_reviews
      invoke  active_record
      remove    db/migrate/20170701174349_create_reviews.rb
blog $
```

(the above code snippet online)

According to the output, the file `db/migrate/20170701174349_create_reviews.rb` has been removed.

List of the Most Important Schema Migration Commands

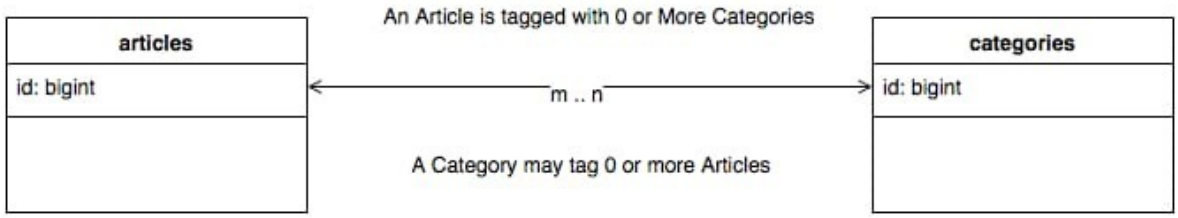
1. `create_table` (also `create_join_table`)
2. `add_column`
3. `remove_column`
4. `rename_column`
5. `change_table`
6. `change_column` (also `change_column_null`, `change_column_default`)
7. `add_index`
8. `remove_index`
9. `add_foreign_key`
10. `remove_foreign_key`

We have already worked with most of them. Let's work with the `create_join_table` command and see how we might want to use the `create_table` command instead.

`create_join_table`

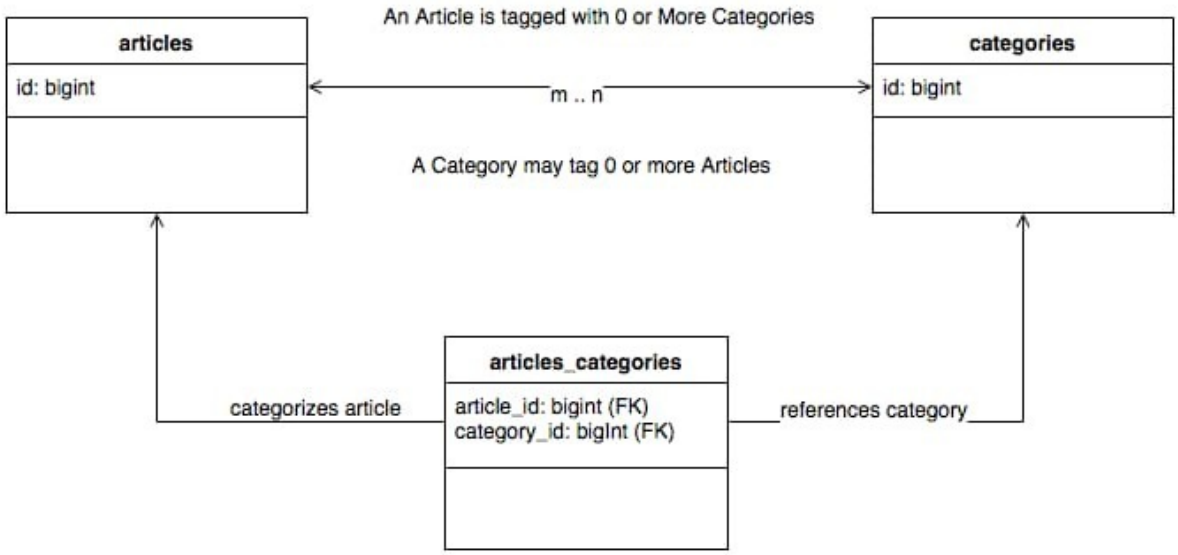
This is used to create a table that implements a many-to-many association between two tables. Although this could have been done with `create_table` as well, the `create_join_table` is a little bit more convenient and makes sure that it follows the Rails conventions.

Let's suppose that we want to enhance our database design with categories for articles. Like this:



Articles And Categories

This is a many-to-many association between Articles and Categories. In order to implement this association at the database level, we will have to introduce an intermediate table:



Articles And Categories With Intermediate Table

In order to implement this new schema, we first need to create the table `categories`. Let's do that:

```
blog $ bin/rails generate migration create_categories
      invoke  active_record
      create    db/migrate/20170701194907_create_categories.rb
blog $
```

(the above code snippet online)

Let's edit the file `db/migrate/20170701194907_create_categories.rb` so that it has the following content:

```
# File: db/migrate/20170701194907_create_categories.rb
#
class CreateCategories < ActiveRecord::Migration[5.1]
  def change
    create_table :categories do |t|
      t.string :name, null: false

      t.timestamps
    end
    add_index :categories, [:name], unique: true, name: 'categories_name_uidx'
  end
end
```

(the above code snippet online)

The above, will create a table named `categories` and the following collumns:

1. an `id` which is going to be the primary key
2. a `name` which is going to be mandatory
3. a `created_at` which will be set to the time the record is created
4. an `updated_at` which will be automatically updated when the record is updated

Let's run this migration:

```
blog $ bin/rake db:migrate
== 20170701194907 CreateCategories: migrating ================================
-- create_table(:categories)
   -> 0.0086s
-- add_index(:categories, [:name], {:unique=>true, :name=>"categories_name_uidx"})
   -> 0.0032s
== 20170701194907 CreateCategories: migrated (0.0119s) =======================

blog $
```

(the above code snippet online)

Great! Table `categories` has been created, and, you can confirm that, by executing the following `psql` command:

```
blog $ psql -d blog_development -c "\d+ categories"
                                                          Table "public.categories"
   Column    |            Type             |                        Modifiers     \
                    | Storage  | Stats target | Description
------------+----------------------------+-------------------------------------\
-------------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('categories_\
id_seq'::regclass) | plain    |              |
 name       | character varying           | not null                            \
                    | extended |              |
 created_at | timestamp without time zone | not null                            \
                    | plain    |              |
 updated_at | timestamp without time zone | not null                            \
                    | plain    |              |
Indexes:
    "categories_pkey" PRIMARY KEY, btree (id)
    "categories_name_uidx" UNIQUE, btree (name)

blog $
```

(the above code snippet online)

Now that we have both `articles` and `categories` table, we can use the `create_join_table` command to create the intermediate table. Let's generate the schema migration first:

```
blog $ bin/rails generate migration create_articles_categories
      invoke  active_record
      create    db/migrate/20170702035400_create_articles_categories.rb
blog $
```

(the above code snippet online)

Now, let's edit the file generated and put the following content inside:

```ruby
# File db/migrate/20170702035400_create_articles_categories
#
class CreateArticlesCategories < ActiveRecord::Migration[5.1]
  def change
    create_join_table :articles, :categories
  end
end
```

(the above code snippet online)

Here, we use the `create_join_table` command that will create a table `articles_categories` and will add two columns inside it:

1. `article_id`

2. `category_id`

Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170702035400 CreateArticlesCategories: migrating ========================
-- create_join_table(:articles, :categories)
   -> 0.0035s
== 20170702035400 CreateArticlesCategories: migrated (0.0036s) ===============

blog $
```

(the above code snippet online)

Right! Migration has been executed. What has happened into our database? Let's run the following command:

```
blog $ psql -d blog_development -c "\d+ articles_categories"
                   Table "public.articles_categories"
   Column    |  Type  | Modifiers | Storage | Stats target | Description
-------------+--------+-----------+---------+--------------+-------------
 article_id  | bigint | not null  | plain   |              |
 category_id | bigint | not null  | plain   |              |

blog $
```

(the above code snippet online)

Two columns, one `article_id` and another one `category_id`.

However, to me, this design does not seem ready.

1. It is missing primary key
2. It is missing a unique index that would combine `article_id` and `category_id`. We wouldn't like to have the same category assigned to the same article more than once.
3. It is missing timestamps, so we don't have any idea when the rows are created or updated.

Let's roll it back.

```
blog $ bin/rake db:rollback
== 20170702035400 CreateArticlesCategories: reverting =========================
-- drop_join_table(:articles, :categories)
   -> 0.0012s
== 20170702035400 CreateArticlesCategories: reverted (0.0042s) ================

blog $
```

(the above code snippet online)

Personally, I prefer to create the intermediate table with proper `create_table` command and introduce all the necessary columns and indexes. Here is my version of `db/migrate/20170702035400_create_articles_categories.rb` file:

```
1  # File db/migrate/20170702035400_create_articles_categories
2  #
3  class CreateArticlesCategories < ActiveRecord::Migration[5.1]
4    def change
5      create_table :articles_categories do |t|
6        t.references :article, foreign_key: {name: 'articles_categories_article_fk'\
7  }, index: false
8        t.references :category, foreign_key: {name: 'articles_categories_category_f\
9  k'}, index: false
10
11       t.timestamps
12     end
13
14     add_index :articles_categories, [:article_id, :category_id], unique: true, na\
15 me: 'articles_categories_article_category_uidx'
16   end
17 end
```

(the above code snippet online)

If you have followed the previous chapters, the above migration is not something new to you. Pay attention how I disable the creation of index for the foreign keys with the `index: false` option. I am doing that because I don't find these indexes necessary in the presence of the index `articles_categories_article_category_uidx` that I introduce on line 12.

When You Want More than The Schema Migration Helpers Can Give You

Sometimes, the schema migration helpers are not enough. In that case, we might want to execute SQL (DDL & DML) commands. We can do that with the method `execute`. You give your custom SQL command as string to the `execute` method.

Migrations and Seed Data

The schema migrations are usually used to define changes in your database schema and are executed once. You can use this feature to carry out data migrations too. This is the recommended practice for production environments, or, in other words, for environments that migrations run only once.

Otherwise, for development or test environments, you might want to use the `bin/rake db:seed` command. This rake task will actually run your `db/seeds.rb` file. If you have database population Ruby commands inside this file, they will be executed.

Closing Schema Migrations

That was a short introduction to Schema Migrations. If you want to read more about Rails Active Record Migrations, have look at the Rails Guides here.

Validations

The validations are very important ingredient of your Models. They make sure that no Model persists in the database with invalid data. The `#save` and `.create` commands trigger the validations and they will not allow anything in the database if the model is not valid. The validations are triggered before the actual database commands take place. They both return `false` if the model is not valid and their bang (!) equivalent raises an `ActiveRecord::RecordInvalid` exception.

See again, for example, the validations for Article:

```
validates :user, presence: true
validates :title, presence: true
validates :text, presence: true
```

(the above code snippet online)

You can see a *presence* validation for `user` attribute of the `Article`. Let's try to create an Article without a user using the `.create!` method. Do this exercise in the rails console for your `blog` project:

```
2.3.3 :043 > Article.create!(title: 'History', text: 'About German History')
   (0.3ms)  BEGIN
   (0.2ms)  ROLLBACK
ActiveRecord::RecordInvalid: Validation failed: User must exist, User can't be bl\
ank
        from (irb):43
2.3.3 :044 >
```

(the above code snippet online)

As you can see nothing has taken place in the database. Do you see the `ROLLBACK`? Also, we see the `ActiveRecord::RecordInvalid` exception that was raised.

Let's see some more details about Model validations.

Active Record Commands that Trigger Validations

Validations are not always triggered. For that reason, you need to be careful when you call methods that persist the state of the object in the database, without calling validations first.

The methods that **do call** validations before executing SQL commands in the database are:

1. `create`
2. `create!`
3. `save`
4. `save!`
5. `update`
6. `update!`

Commands that Skip Validations

On the other hand, the following commands skip validations and should be used with caution:

1. decrement!
2. decrement_counter
3. increment!
4. increment_counter
5. toggle!
6. touch
7. update_all
8. update_attribute
9. update_column
10. update_columns
11. update_counters

Also, the `save` can skip validations if it is given the option `validate: false`.

`valid?` and `invalid?`

Every object instance responds to `valid?` which returns `true` if all the validations are ok. Otherwise, it returns `false`. Note that all validations will be checked.

Let's see an example using the rails console in our blog application:

```
2.3.3 :001 > user = User.new
 => #<User id: nil, email: nil, password_digest: nil, created_at: nil, updated_at\
: nil, email_confirmation_token: nil>
2.3.3 :002 > user.valid?
  User Exists (0.8ms)  SELECT  1 AS one FROM "users" WHERE "users"."email" IS NUL\
L LIMIT $1  [["LIMIT", 1]]
 => false
2.3.3 :003 >
```

(the above code snippet online)

The `user` instance, above, returns `false`, when we call `#valid?`. We can equally call `#invalid?` instead.

```
2.3.3 :003 > user.invalid?
  User Exists (0.4ms)  SELECT  1 AS one FROM "users" WHERE "users"."email" IS NUL\
L LIMIT $1  [["LIMIT", 1]]
 => true
2.3.3 :004 >
```

(the above code snippet online)

but this will return `true`, i.e. the negation of the `#valid?`.

`#errors`

How can we know why an instance is invalid? We can call the `#errors` message. This will return information about the errors that the validation processed has reported.

```
2.3.3 :004 > user.errors
 => #<ActiveModel::Errors:0x007ff73b584168 @base=#<User id: nil, email: nil, pass\
word_digest: nil, created_at: nil, updated_at: nil, email_confirmation_token: nil\
>, @messages={:password=>["can't be blank"], :email=>["can't be blank"]}, @detail\
s={:password=>[{:error=>:blank}], :email=>[{:error=>:blank}]}>
2.3.3 :005 >
```

(the above code snippet online)

Do you see the `messages` `Hash`. For each one of the attributes that was invalid, it contains a corresponding error message.

Note that the `errors` is populated with `messages` only after we trigger validations on the instance, i.e. after we trigger `#valid?` or `#invalid?`. Look at the following rails console interaction that demonstrates that:

```
2.3.3 :005 > user = User.new
 => #<User id: nil, email: nil, password_digest: nil, created_at: nil, updated_at\
: nil, email_confirmation_token: nil>
2.3.3 :006 > user.errors
 => #<ActiveModel::Errors:0x007ff74094fd38 @base=#<User id: nil, email: nil, pass\
word_digest: nil, created_at: nil, updated_at: nil, email_confirmation_token: nil\
>, @messages={}, @details={}>
2.3.3 :007 > user.valid?
  User Exists (0.6ms)  SELECT  1 AS one FROM "users" WHERE "users"."email" IS NUL\
L LIMIT $1  [["LIMIT", 1]]
 => false
2.3.3 :008 > user.errors
 => #<ActiveModel::Errors:0x007ff74094fd38 @base=#<User id: nil, email: nil, pass\
word_digest: nil, created_at: nil, updated_at: nil, email_confirmation_token: nil\
>, @messages={:password=>["can't be blank"], :email=>["can't be blank"]}, @detail\
s={:password=>[{:error=>:blank}], :email=>[{:error=>:blank}]}>
```

(the above code snippet online)

As you can see above, the `errors.messages` is populated only after we trigger the validation process, here done with `#valid?` call.

`:on` Option

Note that validations are triggered when one tries to save an instance in the database. I.e. both when creating a new record and when updating an existing one.

However, if this is not what you want, you can use the `:on` option to tell when the validation should take place. `:on` takes two options: `:create` or `:update`.

So, `validates :name, presence: true, on: :update` runs this presence validation only if the record is updated and not when it is created.

Validation Helpers

In our `blog` application we have worked with a couple of validation helpers:

1. Validation for presence. E.g. in `Article` model `validates :user, presence: true`
2. Validation for uniqueness. E.g. in `User` model `validates :email, uniqueness: { case_sensitive: false }`

However, there are plenty more validations at your disposal.

For example:

Acceptance Validation

This validator can be attached to an attribute such as `:terms_of_service`, and makes sure that the Model is only saved if the value of the attribute is true.

Let's work out an example for this validator. We will add attribute `terms_of_service` to the `User` model and we will make it have the `acceptance` validator switched on.

First things first. Let's add the `terms_of_service` column in the `users` table in the database:

```
blog $ bin/rails generate migration add_terms_of_service_to_users
      invoke  active_record
      create    db/migrate/20170702093324_add_terms_of_service_to_users.rb
blog $
```

(the above code snippet online)

Now, fill in the file `db/migrate/20170702093324_add_terms_of_service_to_users.rb` with the following content:

```ruby
# File: db/migrate/20170702093324_add_terms_of_service_to_users.rb
#
class AddTermsOfServiceToUsers < ActiveRecord::Migration[5.1]
  def change
    add_column :users, :terms_of_service, :boolean, default: false, null: false

    reversible do |direction|
      direction.up do
        say_with_time('Setting terms of service to true for all users') do
          User.reset_column_information
          User.find_each do |user|
            user.terms_of_service = true
            user.save!
          end
        end
      end
    end
  end
end
```

(the above code snippet online)

We are using the `add_column` Active Record Migration method. This will add the column named `terms_of_service` to the `users` table. Also, it will assume the default value being `false` and will not allow any null values. Also, we make sure that existing users have accepted the terms of service by setting their `terms_of_service` value to `true`. Note that I am using the `#save!` version of `#save` to make sure that I still have a valid `User` instance. If this is not the case, all the migration will be rolled back.

Hint: Actually, the migration is rolled back, in case of a failure, if the database management system supports *transactions* at the data definition level, which is the case with PostgreSQL, but not with MySQL.

Information: See also how we use the `reversible` to tell Rails that if this migration is rolled back then there is no code that corresponds to the setting of the column `true` for all the users. You can try both migration and rollback, if you want to see how this code behaves.

Let's run the migration now:

```
blog $ bin/rake db:migrate
== 20170702093324 AddTermsOfServiceToUsers: migrating ========================
-- add_column(:users, :terms_of_service, :boolean, {:default=>false, :null=>false\
})
   -> 0.0092s
-- Setting terms of service to true for all users
   -> 0.0574s
== 20170702093324 AddTermsOfServiceToUsers: migrated (0.0668s) ================

blog $
```

(the above code snippet online)

Nice! The migration has been executed successfully. Let's confirm what we have done with the help of `psql`.

```
blog $ psql -d blog_development -c "select * from users;"
 id |           email            |                       password_digest         \
                |         created_at         |         updated_at         | email_\
confirmation_token | terms_of_service
----+----------------------------+-----------------------------------------------\
---------------+----------------------------+----------------------------+-------\
-------------------+------------------
  8 | foo@gmail.com              | $2a$10$WuL5ic/qRTL2.AK9vAIDxuoHZAf7V/dFdlLzFpg\
Xdnwc1GhtrJj8i | 2017-06-04 07:25:59.681835 | 2017-07-02 09:52:46.964696 | 3sRJUZ\
vDS0wNMNY2pfbU_A   | t
  9 | bar@gmail.com              | $2a$10$g/4AILw8UXB1s8G01M8P1efgzs0iZfNfw.BxdHW\
2HNIxMhgkgqrr2 | 2017-06-04 13:51:46.423347 | 2017-07-02 09:52:46.96815  |       \
                   | t
 10 | mary@gmail.com             | $2a$10$YLIJQ0AEVCTFnDgYs013j.J32Hk9df2YPuvouEC\
x.TjrttueV9wVq | 2017-06-10 06:46:05.276602 | 2017-07-02 09:52:46.970002 | yStnEr\
kVCFC9mYNHKFNYdA   | t
 12 | john@gmail.com             | $2a$10$.JIoJKajyd2abxCkYFE0O.xaSyPLQ95ylXbOlyM\
DvdlKUPKv/l5Mi | 2017-06-10 08:02:26.555474 | 2017-07-02 09:52:46.971858 |       \
                   | t
```

```
 13 | peter@gmail.com            | $2a$10$MGPCmf92M0kOC78Xm4zmxOAOQ0jEFWlbjCRGXpN\
Fv6s3jNIdNKFeu | 2017-06-11 05:10:39.504549 | 2017-07-02 09:52:46.973647 |       \
                    | t
 16 | paul@gmail.com             | $2a$10$Ba9KZ9pGgiFuc5o8U8TDX.vv6SbLACUVrS89k7Y\
o8/dCtJm4TOIwS | 2017-06-11 06:36:41.545059 | 2017-07-02 09:52:46.975448 | 0IRMYX\
9gcpVOtmn28kGLYw   | t
 17 | kristin@gmail.com          | $2a$10$COrxjcEMPOh/t3u.Hf5uhuW.NaQ4C79EzGB.s0r\
GHFTbIn6kQTaLe | 2017-06-11 06:55:22.171308 | 2017-07-02 09:52:46.977246 | nVBfAi\
fMQBaRh86B9dzWMg   | t
 18 | angela@gmail.com           | $2a$10$HXoMZTFWqPqxobuUFI7yRuNd83Q.xoyvm1MXSHT\
6PBwJGwZbYCJ/C | 2017-06-11 06:58:18.846746 | 2017-07-02 09:52:46.979059 |       \
                    | t
 19 | peter2@gmail.com           | $2a$10$JFvKqKhWBIq/2bXMNlAwa.ENtE653CCoMEPPNgN\
kgVK7G.OIy2JVK | 2017-06-11 07:11:28.007321 | 2017-07-02 09:52:46.98084  | oEiLZ_\
g3t79GvpUb9XfZAg   | t
 20 | john2@gmail.com            | $2a$10$qVeJe/d3MX7tcdtC/aZsUOt1cK.TNeAKJyilU87\
HeP0mQHZFnIbOe | 2017-06-11 07:11:56.728099 | 2017-07-02 09:52:46.982927 | 5bW4t7\
07RefeArX5jRBLlQ   | t
 21 | kelly@gmail.com            | $2a$10$7ajlqQuSh/ZkMuVulje9B.deuNVX0EGZsDSm0kW\
Fx8Du0cFnQ4dgS | 2017-06-11 09:45:22.402168 | 2017-07-02 09:52:46.984812 |       \
                    | t
 24 | foo-bar@gmail.com          | $2a$10$aCjEQ6n8IbaBuO27oy47I.JPN4GGSxvmW33yicM\
Dz2z1CX5bqROYG | 2017-06-20 12:02:35.390415 | 2017-07-02 09:52:46.986605 | w-0oSE\
HckGCyLXRUQB5cHg   | t
 25 | foo-201706221324@gmail.com | $2a$10$uiZZPzx2l.mtN6iRBtzfT.q4LnxZu4Xq8yOaVzm\
NpYAuXhCTH6LPe | 2017-06-22 10:24:38.300964 | 2017-07-02 09:52:46.988666 | iGrDo8\
1mfyzE_tHWGslrMQ   | t
(13 rows)

blog $
```

(the above code snippet online)

Do you see the `terms_of_service` column at the far right? Perfect! All existing users have value `t` for this column.

Let's now enhance our `User` model to have the *acceptance* validator attached to the `terms_of_service`.

```ruby
# File app/models/user.rb
#
class User < ApplicationRecord
  has_many :articles

  has_secure_password

  validates :email, presence: true, uniqueness: { case_sensitive: false }
  validates :terms_of_service, acceptance: true

...
end
```

(the above code snippet online)

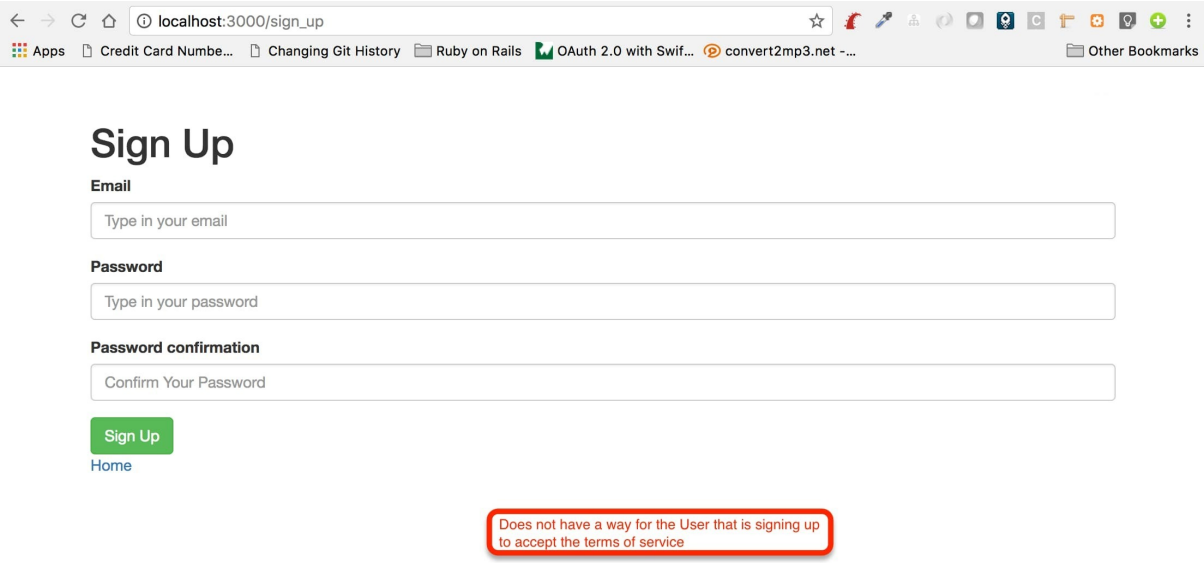
It is the line 9 that we have added. We now tell that a `User` is only valid if the terms of service have been accepted.

If we run the tests that we have developed for our `blog` application, we will now see a lot of failures:

```
blog $ bin/rails test
Run options: --seed 11689

# Running:

E

Error:
ArticleTest#test_should_be_invalid_without_a_text:
ActiveRecord::RecordInvalid: Validation failed: Terms of service must be accepted
    test/models/article_test.rb:19:in `block in <class:ArticleTest>'


bin/rails test test/models/article_test.rb:18
...

Finished in 0.793411s, 22.6869 runs/s, 44.1133 assertions/s.
18 runs, 35 assertions, 2 failures, 6 errors, 0 skips
blog $
```

(the above code snippet online)

These tests fail, probably, because of this new feature that we have added. And this is because the sign up form, does not have a way for the user to accept the terms of service.



Sign Up Form Does Not Have a Way for User to Accept Terms Of Service

Let's enhance our form to allow for the user to accept the terms of service. Here is the new version of the file `app/views/sign_up/new.html.haml`:

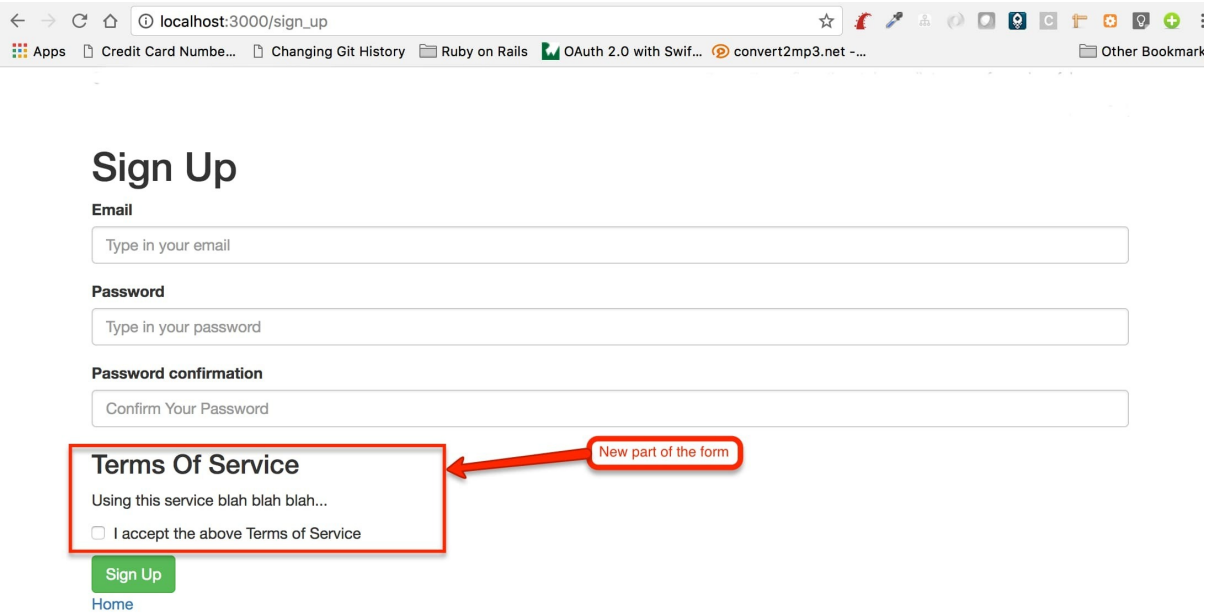
```
- # File: app/views/sign_up/new.html.haml
- #
%h1 Sign Up

= form_for @user, url: sign_up_path, method: :post do |f|

  = render partial: 'shared/errors', locals: {model: @user, explanation: 'prohibi\
ted User from signing up:'}

  .form-group
    = f.label :email
    = f.email_field :email, placeholder: 'Type in your email', class: 'form-contr\
ol'

  .form-group
    = f.label :password
    = f.password_field :password, placeholder: 'Type in your password', class: 'f\
orm-control'

  .form-group
    = f.label :password_confirmation
    = f.password_field :password_confirmation, placeholder: 'Confirm Your Passwor\
```

```
23 d', class: 'form-control'
24
25   .terms-of-service
26     %h3 Terms Of Service
27     %p
28       Using this service blah blah blah...
29
30   .checkbox
31     = f.label :terms_of_service do
32       = f.check_box :terms_of_service
33       I accept the above Terms of Service
34
35   = f.submit 'Sign Up', class: 'btn btn-success'
36
37 %div
38   = link_to 'Home', root_path
```

(the above code snippet online)

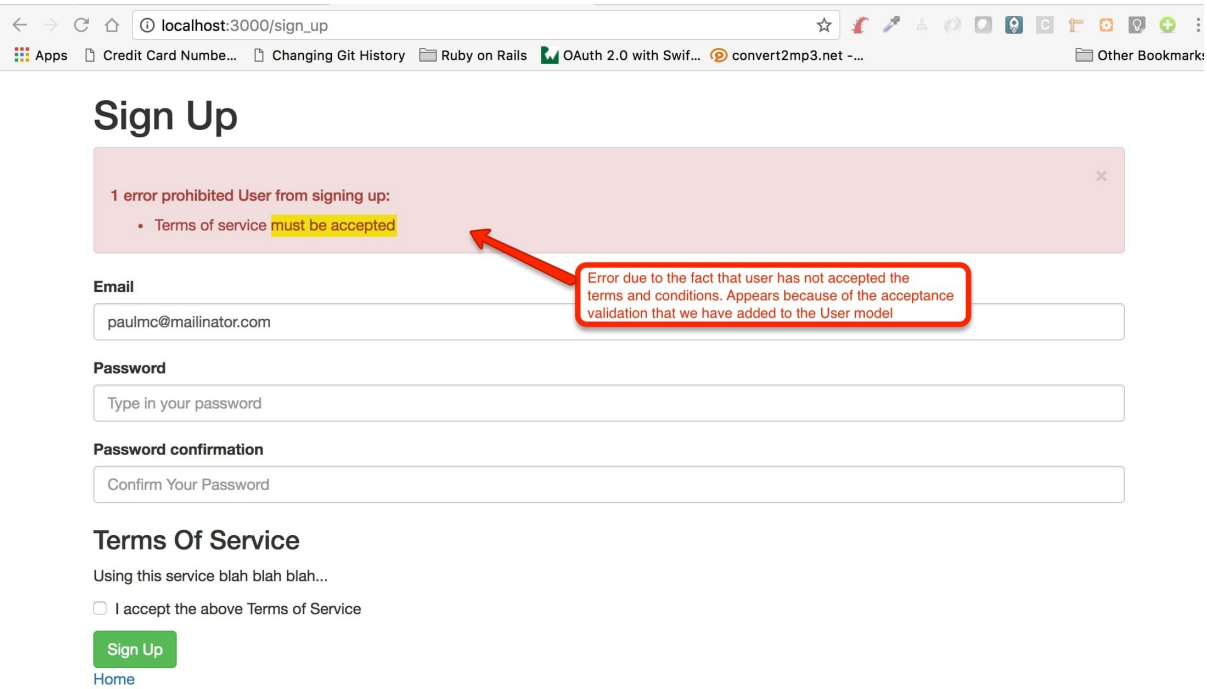
The new lines are the lines 21 to 29. We display a message that is supposed to be the actual terms of service and then we display a checkbox that needs to be checked by the user. This checkbox is bound to the `terms_of_service` property.

If you visit the new form here: http://localhost:3000/sign_up, you will see this:



Sign Up Form With Terms Of Service

If you try to submit the Sign Up without checking the box for terms of service acceptance, you will see this:



Sign Up Form - Error Due To Terms Not Accepted

The validation of `acceptance` takes place and the error message is `must be accepted`.

Let's see what happens when the user accepts the terms and conditions. Fill in the password fields again, check the box for terms of service and resubmit.

Hmmmm! You will see the same error appearing again. What are we missing here?

If we look at the server-side logs, we will see something like this:

```
Started POST "/sign_up" for 127.0.0.1 at 2017-07-02 14:39:33 +0300
Processing by SignUpController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"Z4PzO0YIvhnK3GfykjGMuQzrMbiNcH\
GzTwAb9HJis9AOUkZah8/ce0kg0BYrxaP8PKZST+e47tKSRMB9IXzAeQ==", "user"=>{"email"=>"p\
aulmc@mailinator.com", "password"=>"[FILTERED]", "password_confirmation"=>"[FILTE\
RED]", "terms_of_service"=>"1"}, "commit"=>"Sign Up"}
  User Load (0.3ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS NULL LI\
MIT $1  [["LIMIT", 1]]
Unpermitted parameter: :terms_of_service
   (0.1ms)  BEGIN
  User Exists (0.3ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "paulmc@mailinator.com"], ["LIMIT", 1]]
   (0.1ms)  ROLLBACK
  Rendering sign_up/new.html.haml within layouts/application
  Rendered shared/_errors.html.haml (3.9ms) [cache miss]
  Rendered sign_up/new.html.haml within layouts/application (11.0ms)
  Rendered layouts/_flash.html.haml (3.8ms) [cache miss]
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS NULL LI\
```

```
19 MIT $1  [["LIMIT", 1]]
20   Rendered layouts/_signed_in_user.html.haml (4.4ms) [cache miss]
21 Completed 200 OK in 153ms (Views: 85.9ms | ActiveRecord: 1.3ms)
```

(the above code snippet online)

Do you see the line 5? `Unpermitted parameter: :terms_of_service`. This error is raised when `SignUpController#create` action is trying to parse the parameters submitted by the form. The actual piece of code that parses the parameters is this:

```
1 def user_params
2   params.require(:user).permit(:email, :password, :password_confirmation)
3 end
```

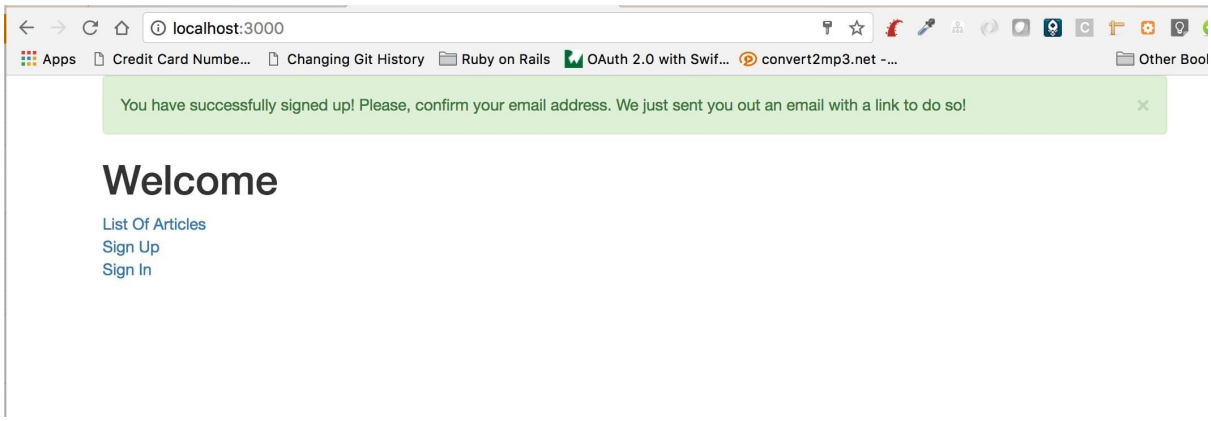
(the above code snippet online)

As you can read in this piece of code, the `:terms_of_service` is not included in the list of permitted parameters. Let's amend this line:

```
1 params.require(:user).permit(:email, :password, :password_confirmation, :terms_of\
2 _service)
```

(the above code snippet online)

And now, let's refill the Sign Up form, checking the terms of service box again, and resubmit.



Sign Up Was Successful With Terms Of Service Included

This fixes the problem with the sign up form.

However, we still get errors in our tests. This is because the factories that we use for users do not build valid users any more.

This is what we have:

```ruby
# File: test/factories/users.rb
#
FactoryGirl.define do
  factory :user do
    email { "#{SecureRandom.hex}@mailinator.com" }
    password '12345678'
  end
end
```

(the above code snippet online)

We are missing the `terms_of_service`. Let's amend the above file to cater for `terms_of_service` too:

```ruby
# File: test/factories/users.rb
#
FactoryGirl.define do
  factory :user do
    email { "#{SecureRandom.hex}@mailinator.com" }
    password '12345678'
    terms_of_service true
  end
end
```

(the above code snippet online)

If you run your tests now, you only get 2 failures. They are the integration tests inside the file `test/integration/sign_up_test.rb`. If you read the code, you will see that the test does not send the `terms_of_service` as part of the `params`:

```ruby
post sign_up_path, params: {user: {email: email, password: test_password, passwor\
d_confirmation: test_password}}
```

(the above code snippet online)

We need to amend the above line as follows:

```ruby
post sign_up_path, params: {user: {email: email, password: test_password, passwor\
d_confirmation: test_password, terms_of_service: true}}
```

(the above code snippet online)

This needs to be done for both tests because they both use the same `post` method call.

Now, if you run your tests, they will be green:

```
blog $ bin/rails test
Running via Spring preloader in process 23119
Run options: --seed 61821

# Running:

..................

Finished in 0.709230s, 25.3796 runs/s, 74.7289 assertions/s.
18 runs, 53 assertions, 0 failures, 0 errors, 0 skips
blog $
```

(the above code snippet online)

But don't forget to run your system tests too, because they do not run by default:

If you do that, you will see that the system test fails:

```
blog $ bin/rails test:system
Run options: --seed 23060

# Running:

Puma starting in single mode...
* Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
* Min threads: 0, max threads: 1
* Environment: test
* Listening on tcp://0.0.0.0:51536
Use Ctrl-C to stop
[Screenshot]: tmp/screenshots/failures_test_sign_up_happy_path.jpg

F

Failure:
SignUpTest#test_sign_up_happy_path [test/system/sign_up_test.rb:20]:
expected to find css ".alert.alert-success" with text "You have successfully sign\
ed up! Please, confirm your email address. We just sent you out an email with a l\
ink to do so!" but there were no matches


bin/rails test test/system/sign_up_test.rb:6

Finished in 3.576432s, 0.2796 runs/s, 0.2796 assertions/s.
1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
blog $
```

(the above code snippet online)

Obviously, this is because of the reason that the test does not simulate the acceptance of the terms of service. This is the new version of the test that makes sure we check the terms of service box:

```
1  # File: test/system/sign_up_test.rb
2  #
3  require 'application_system_test_case'
4
5  class SignUpTest < ApplicationSystemTestCase
6    test 'sign up happy path' do
7      email = "#{SecureRandom.hex}@mailinator.com"
8      test_password = '12345678'
9
10     visit root_url
11
12     click_link 'Sign Up'
13
14     fill_in 'Email', with: email
15     fill_in 'Password', with: test_password
16     fill_in 'Password confirmation', with: test_password
17     check 'I accept the above Terms of Service'
18
19     click_button 'Sign Up'
20
21     assert_selector '.alert.alert-success', text: 'You have successfully signed u\
22 p! Please, confirm your email address. We just sent you out an email with a link \
23 to do so!'
24   end
25 end
```

(the above code snippet online)

Line 17 is the only new line that we have added. It is using the Capybara API to check a checkbox.

If you now run your system tests, they will succeed:

```
1  blog $ bin/rails test:system
2  Run options: --seed 27312
3
4  # Running:
5
6  Puma starting in single mode...
7  * Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
8  * Min threads: 0, max threads: 1
9  * Environment: test
10 * Listening on tcp://0.0.0.0:51671
11 Use Ctrl-C to stop
```

```
.

Finished in 1.954052s, 0.5118 runs/s, 0.5118 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
blog $
```

(the above code snippet online)

Bingo!

Testing with `RSpec`, will require you to do the similar changes. Please, go ahead and do them.

Testing with `Cucumber`. Your `sign_up.feature` that tests the Sign Up feature needs to be adjusted. Please, go ahead and do it.

That was a long exercise on the acceptance validation.

Let's continue with the rest of the validations.

Confirmation

We have already used this validation when we used `has_secure_password` for the `User` model. It is used to tell Rails that we want Model to have an attribute confirming the value of another one. For example:

```
validates :email, confirmation: true
```

(the above code snippet online)

will define a virtual attribute named `:email_confirmation` that, if given, should have the same value like the `:email` attribute.

This is what `has_secure_password` does behind the scenes:

```
validates :password, confirmation: true
```

(the above code snippet online)

and this is the reason we used the `:password_confirmation` as an attribute for Signing Up.

Exclusion

It is used to tell which values an attribute is not allowed to take.

Format

This validation makes sure that the value of the attributes matches a regular expression.

Let's see that with the following example. We will add this validation for the `:email` attribute for the `User` model:

```
validates :email, format: {with: /\A\S+@\S+\.\S+\z/, message: 'is not valid email\
'}
```

(the above code snippet online)

Add the above line to your `User` model. The `with: /\A\S+@\S+\.\S+\z/` specifies the regular expression against which the email value given is going to be matched. If the email given does not match this regular expression, then the error message `is not valid email` will be attached to the `:email` attribute of the `User` instance.

> *Hint:* The above regular expression makes sure that value for `:email` contains some non-white space characters before the symbol `@`, then some more non-white space characters, followed by a `.` and then some more non-white space characters.

With the above format validation in place, you will encounter things like these when you will try to give an invalid email:

Invalid Error Validation

> *Hint:* Because of the fact that the field is declared of type `email`, your browser already does some validation. So you may not be able to submit some possibly invalid values.

The example for the format validator that we saw above also takes advantage of the fact that we can specify the `message` that can be attached to a validation error. This is applicable to all validators.

Inclusion

The inclusion validation is used when we want the values of an attribute to be limited to a specific set of values. For example:

```
validates :size, inclusion: {in: %w(small medium large extra_large)}
```

(the above code snippet online)

Length

This checks a string value for its length. For example, we can specify that the password should be at least 8 characters long:

```
validates :password, length: { minimum: 8 }
```

(the above code snippet online)

Or we can also specify a maximum limit with the `maximum` key.

Sometimes we may want to specify an exact length value. In that case we use the key `is`, like in the following example:

```
validates :account_number, length: { is: 10 }
```

(the above code snippet online)

Numericality

This validator makes sure that we save values that are numbers. For example:

```
validates :salary, numericality: true
```

(the above code snippet online)

If you want to make sure that values are integers, you can set `only_integer: true`. For example:

```
validates :age, numericality: { only_integer: true }
```

(the above code snippet online)

And if you want to be more specific, you can use the following keys:

1. `greater_than`
2. `greater_than_or_equal_to`
3. `equal_to`
4. `less_than`
5. `less_than_or_equal_to`
6. `other_than`
7. `odd`
8. `even`

Presence

We have already seen many examples of the presence validator, which makes sure that an attribute has a non-nil value. Also, if the value is a string, blank strings are not allowed either.

Absence

This makes sure that the value is either `nil` or a blank string. This is the reverse of the presence validator and internally uses the `!present?` to find out whether something is absent or not. In other words, absence means *not-presence* here.

> *Note for both `presence` and `absence`:* Since both rely on `present?`, you need to be careful with the boolean values. The boolean value `false` is not considered present, i.e. it is considered absent. Use `inclusion` or `exclusion` validators against the set `%w(true false)` to deal with the presence or absence of attributes that take boolean values.

Uniqueness

We have already worked with the uniqueness validator in the `User` model, for the attribute `:email`.

`validates_with` and `validates_each`

These are tools for custom validations. You can read more about them in the Rails Guides in Active Record Validations Chapter.

Validation Common Options

There are some common options for all the validators:

1. `allow_nil`. This can be used when we want to allow `nil` value for an attribute that otherwise takes values from a restricted set.
2. `allow_blank`. This allows both `nil` and blank strings.
3. `message`. It is used to customize the error message attached to an attribute when validation fails.
4. `on`, which takes one of the values `:create` or `:update`, in order to limit validations on creation or on update, respectively.

Conditional Validations

We can tell that a validation is conditional with the help of the keys `:if` or `:unless`. For example:

```ruby
class Order < ApplicationRecord
  validates :card_number, presence: true, if: :paid_with_card?

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

(the above code snippet online)

You can see that the presence validator above has a condition attached: `if: :paid_with_card?`. The good thing with that is that the condition is evaluated every time the validation takes place. Here, the value to the key `:if` is the symbol of a method that should return a truethy value, when called.

Read more about the Conditional Validations in the Rails Guides here.

Custom Validations

We have already created a custom validation. See the line `validate :authenticate` in the `SignIn` model and the method implementation:

```ruby
  def authenticate
    return unless email.present? && password.present?

    @user = User.find_by_email(email.downcase).try(:authenticate, password)

    unless @user
      error_message = 'Wrong Credentials!'
      errors[:email] << error_message
      errors[:password] << error_message
    end
  end
```

(the above code snippet online)

You can read about other ways one can create custom validations here.. And you can read more about the errors handling here.

Active Record Callbacks

We have already seen an example of an Active Record Callback. Look at the code in `User` model in our `blog` application. You can see the line of code `before_create :generate_email_confirmation_token`. This is a callback for the *before create* event.

List of Available Callbacks

Callbacks are being called when you *create*, *update* or *delete* an object and on some other cases.

Callbacks On Create

1. `before_validation`
2. `after_validation`
3. `before_save`
4. `around_save`
5. `before_create`
6. `around_create`
7. `after_create`
8. `after_save`
9. `after_commit/after_rollback`

Callbacks on Update

1. `before_validation`
2. `after_validation`
3. `before_save`
4. `around_save`
5. `before_update`
6. `around_update`
7. `after_update`
8. `after_save`
9. `after_commit/after_rollback`

Callbacks On Deletion

1. `before_destroy`
2. `around_destroy`
3. `after_destroy`
4. `after_commit/after_rollback`

Callbacks are being called in the sequence that they are given above. For example, when you create a new object, the first callback called is the `before_validation` and the second one is the `after_validation`, whereas the last one is the `after_commit` (if everything goes ok) or the `after_rollback` (if things cannot be committed, but should be rolled back instead).

`after_initialize`

The `after_initialize` is called whenever an Active Record is initialized either by just calling `.new` or when the instance is created as a result of fetching an existing record from the database.

Let's carry out an example here. Amend the `User` model so that it has the following code:

```ruby
# File app/models/user.rb
#
class User < ApplicationRecord
  has_many :articles

...

  after_initialize :user_initialized

...

  private

...

  def user_initialized
    puts "User #{email} is initialized"
  end
end
```

(the above code snippet online)

We have added line 14 and lines 35 to 37. The method `user_initialized` will be called whenever we initialize an instance of this class. Start your rails console and issue two commands. First: `User.new` and then `User.first` (assuming that you have some users in your database). You will see something like this:

```
2.3.3 :001 > User.new
User  is initialized
 => #<User id: nil, email: nil, password_digest: nil, created_at: nil, updated_at\
: nil, email_confirmation_token: nil, terms_of_service: false>
2.3.3 :002 > User.first
  User Load (0.6ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
IT $1  [["LIMIT", 1]]
User foo@gmail.com is initialized
 => #<User id: 8, email: "foo@gmail.com", password_digest: "$2a$10$WuL5ic/qRTL2.A\
K9vAIDxuoHZAf7V/dFdlLzFpgXdnw...", created_at: "2017-06-04 07:25:59", updated_at:\
 "2017-07-02 09:52:46", email_confirmation_token: "3sRJUZvDS0wNMNY2pfbU_A", terms\
_of_service: true>
2.3.3 :003 >
```

(the above code snippet online)

Do you see the two lines `User .... is initialized`. They have been printed because of the `after_initialize` hook.

Note that `after_initialize` is a good tool for you to do things on top of Active Record initializer without having to override the Active Record initializer.

after_find

The `after_find` callback is called whenever a record is fetched from the database. It is called before `after_initialized`, if both have been defined.

Let's add an `after_find` hook in our `User` model:

```
1  # File app/models/user.rb
2  #
3  class User < ApplicationRecord
4    has_many :articles
5  ...
6    after_initialize :user_initialized
7    after_find :user_found
8
9  ...
10
11   private
12
13 ...
14
15   def user_initialized
16     puts "User #{email} is initialized"
17   end
18
19   def user_found
20     puts "User #{email} found"
21   end
22 end
```

(the above code snippet online)

We have added line 15 and lines 40 to 42. Now, let's do the same test with the rails console, to initialize and fetch a record (make sure that you restart your rails console session so that it takes into account your changes)

```
1  2.3.3 :006 >   User.new
2  User  is initialized
3   => #<User id: nil, email: nil, password_digest: nil, created_at: nil, updated_at\
4  : nil, email_confirmation_token: nil, terms_of_service: false>
5  2.3.3 :007 > User.first
6    User Load (0.5ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
7  IT $1  [["LIMIT", 1]]
8  User foo@gmail.com found
9  User foo@gmail.com is initialized
10  => #<User id: 8, email: "foo@gmail.com", password_digest: "$2a$10$WuL5ic/qRTL2.A\
11 K9vAIDxuoHZAf7V/dFdlLzFpgXdnw...", created_at: "2017-06-04 07:25:59", updated_at:\
```

```
 "2017-07-02 09:52:46", email_confirmation_token: "3sRJUZvDS0wNMNY2pfbU_A", terms\
_of_service: true>
```

(the above code snippet online)

You can see that, first, the `after_find` hook was called, and then the `after_initialize` hook.

Cool!

after_touch

The `after_touch` is called when an Active Record is touched. Whenever you call `#touch` on an Active Record its `updated_at` attribute gets updated. Also, the hook `after_touch` is called.

Let's see that with the `User` model:

```
# File app/models/user.rb
#
class User < ApplicationRecord
...
  after_find :user_found
  after_touch :user_touched
...

  private

...
  def user_found
    puts "User #{email} found"
  end

  def user_touched
    puts "User #{email} touched"
  end
end
```

(the above code snippet online)

We have added lines 16 and 45 to 47. Restart your rails console and try the following:

```
2.3.3 :001 > u = User.first
  User Load (1.2ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
IT $1  [["LIMIT", 1]]
User foo@gmail.com found
User foo@gmail.com is initialized
 => #<User id: 8, email: "foo@gmail.com", password_digest: "$2a$10$WuL5ic/qRTL2.A\
K9vAIDxuoHZAf7V/dFdlLzFpgXdnw...", created_at: "2017-06-04 07:25:59", updated_at:\
 "2017-07-02 09:52:46", email_confirmation_token: "3sRJUZvDS0wNMNY2pfbU_A", terms\
_of_service: true>
2.3.3 :002 > puts "Updated: #{u.updated_at}"
Updated: 2017-07-02 09:52:46 UTC
 => nil
2.3.3 :003 > u.touch
   (0.2ms)  BEGIN
  SQL (3.0ms)  UPDATE "users" SET "updated_at" = '2017-07-03 16:22:37.504166' WHE\
RE "users"."id" = $1  [["id", 8]]
User foo@gmail.com touched
   (0.7ms)  COMMIT
 => true
2.3.3 :004 > puts "Updated: #{u.updated_at}"
Updated: 2017-07-03 16:22:37 UTC
 => nil
2.3.3 :005 >
```

(the above code snippet online)

- On command 001, we fetch the first user. You can see the `after_find` and `after_initialize` hooks being fired.
- On command 002, we print the `updated_at` value of the User at hand.
- On command 003, we touch the User at hand and we see the `after_touch` hook being fired.
- On command 004, we print the `updated_at` value again. You can see how it now contains a new value.

Skipping Callbacks

You may want to write some code for the state of an Active Record but you wouldn't like to execute the callbacks attached to the model. The following methods do not execute the callbacks:

1. `decrement`
2. `decrement_counter`
3. `delete`
4. `delete_all`
5. `increment`
6. `increment_counter`
7. `toggle`
8. `touch`

9. `update_column`
10. `update_columns`
11. `update_all`
12. `update_counters`

You should be using these methods without caution and making sure that by-passing the callbacks is really what you want to do. Sometimes, for example, we want to carry out an update in the database without calling any of the callbacks. In that case, these methods here are very useful.

Halting Execution

The callbacks are executed one by one in predefined order and according to the order they have been registered. Everything is executed inside a big transaction. If any of the callbacks raises an exception, the execution chain gets halted and a ROLLBACK is issued, and nothing is committed to the database.

Note that if you want to intentionally stop a chain, from within inside of your callback implementations, you can call `throw :abort`.

Relational Callback

There is an option for the `has_many` (or `has_one`) association that allows you to register a callback that has to do with the relation of objects.

For example, look at `User` model:

```
has_many :articles
```

(the above code snippet online)

We can turn this to this:

```
has_many :articles, dependent: :destroy
```

(the above code snippet online)

With the `dependent: :destroy` option we create an `after_destroy` callback for the `User` model that deletes the Articles that belong to the `User` that's being deleted.

Go ahead and do this change to your `User` model. Then start your rails console again:

```
2.3.3 :001 > u = Article.first.user
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
"id" ASC LIMIT $1  [["LIMIT", 1]]
  User Load (0.5ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = $1 LIMIT\
 $2  [["id", 8], ["LIMIT", 1]]
User foo@gmail.com found
User foo@gmail.com is initialized
 => #<User id: 8, email: "foo@gmail.com", password_digest: "$2a$10$WuL5ic/qRTL2.A\
K9vAIDxuoHZAf7V/dFdlLzFpgXdnw...", created_at: "2017-06-04 07:25:59", updated_at:\
 "2017-07-03 16:22:37", email_confirmation_token: "3sRJUZvDS0wNMNY2pfbU_A", terms\
_of_service: true>
2.3.3 :002 > u.articles.count
   (0.6ms)  SELECT COUNT(*) FROM "articles" WHERE "articles"."user_id" = $1  [["u\
ser_id", 8]]
 => 7
2.3.3 :003 > u.destroy
   (0.2ms)  BEGIN
  Article Load (0.3ms)  SELECT "articles".* FROM "articles" WHERE "articles"."use\
r_id" = $1  [["user_id", 8]]
  SQL (1.0ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 31]]
  SQL (0.3ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 32]]
  SQL (0.3ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 33]]
  SQL (0.3ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 34]]
  SQL (0.3ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 35]]
  SQL (0.3ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 36]]
  SQL (0.2ms)  DELETE FROM "articles" WHERE "articles"."id" = $1  [["id", 37]]
  SQL (0.4ms)  DELETE FROM "users" WHERE "users"."id" = $1  [["id", 8]]
   (2.0ms)  COMMIT
 => #<User id: 8, email: "foo@gmail.com", password_digest: "$2a$10$WuL5ic/qRTL2.A\
K9vAIDxuoHZAf7V/dFdlLzFpgXdnw...", created_at: "2017-06-04 07:25:59", updated_at:\
 "2017-07-03 16:22:37", email_confirmation_token: "3sRJUZvDS0wNMNY2pfbU_A", terms\
_of_service: true>
2.3.3 :004 >
```

(the above code snippet online)

With the `has_many :articles, dependent: :destroy` we have managed to delete all the articles of the user at hand:

- On command 001, we fetch a User that has Articles.
- On command 002, we count the number of Articles this User has.
- On command 003, we delete the User. As you can see from the SQL commands, Active Record first deletes the Articles of the User before deleting the User itself.

Conditional Callbacks

Like with validations, you can specify conditions under which a callback will be called or conditions under which the callback will not be called. These are the `:if` and `:unless` options at

the callback registration point.

Read about this here

Transactional Callbacks

The `after_commit` and `after_rollback` hooks are called after everything commits to the database or after a rollback takes place, respectively. Note that some times we want to write hooks for these events but only for a specific case among *create*, *update* or *delete*. In that case, we can use the option `:on`. For example:

```
after_commit :send_email, on: :create
```

(the above code snippet online)

will call the `send_email` method only after create commit takes place, but not after update or destroy.

Closing On Callbacks

If you want to read more about Active Record Callbacks, you can read the Rails Guides here.

> *Task:* Remove the `after_initialize`, `after_find` and `after_touch` callbacks from your `User` model.

Active Record Associations

We have already seen how to associate one object to another. For example, we have used the method `has_many` to associate a User to many Articles. And the `belongs_to`, in order to associate the Article back to the User that the Article belongs to. That was one way to implement a one-to-many association.

You could have also used the method `has_one` instead, if the association was one-to-one.

Note that the `belongs_to` method should be placed in the Model that has the foreign key, whereas the `has_many` or `has_one` should be placed in the other Model. That's why we have put the `belongs_to` inside the Article model. It is the `articles` table that has the `user_id` as foreign key to `users` table.

Declaring Associations Introduces a lot of Methods

Declaring an association from one model to another, defines a set of methods in your class.

For example, declaring the `belongs_to :user` in the `Article` model, you get for free the following:

1. `user(force_reload = false)`. Returns the associated user. `nil` is returned if none is found.
2. `user = another_user`. Allows you to assign another user to the article at hand.

3. `build_user(attributes = {})`. Returns a new `User` object that has been instantiated with `attributes` and linked to the `Article` at hand, through a foreign key, but has not yet been saved.
4. `create_user(attributes = {})`. Returns a new `User` object that has been instantiated with `attributes`, linked to the `Article` at hand, through a foreign key, and that has already been saved (if it passed the validation).
5. `create_user!(attributes = {})`. Does the same as `#create_association`, but raises `ActiveRecord::RecordInvalid`, if the record is invalid.

Equivalent things take place when you declare a `has_many` association.

It is good to start getting yourself acquainted with the documentation comments inside the file here and maybe here

Remember: Ruby on Rails is an open source project and its source code can be found here.

`:has_many :through` Association

Let's see another view on `has_many` association. Do you remember that we created the `articles_categories` table to support the many-to-many relationship between Articles and Categories?

Let's now enhance our models with the correct association methods.

Model for Categories

We will need the model for the Categories. This is a first simple version.

```
# File: app/models/category.rb
#
class Category < ApplicationRecord
  validates :name, presence: true, uniqueness: {case_sensitive: false}
end
```

(the above code snippet online)

Model for Articles Categories Association Table

We also need a model for the `articles_categories` table:

```
# File: app/models/articles_category.rb
#
class ArticlesCategory < ApplicationRecord
  belongs_to :article
  belongs_to :category
end
```

(the above code snippet online)

The `ArticlesCategory` model needs to have the `belongs_to` parts because the table `articles_categories` has the foreign keys to the other tables.

`has_many` in Article

The `Article` needs to be enhanced with a `has_many` association, corresponding to the `belongs_to :article` inside the `ArticleCategory`:

```
1  # File: app/models/article.rb
2  #
3  class Article < ApplicationRecord
4    belongs_to :user
5    has_many :articles_categories, dependent: :destroy
6
7    validates :user, presence: true
8    validates :title, presence: true
9    validates :text, presence: true
10 end
```

(the above code snippet online)

See line 5. It is the line that we have added. Note the `dependent: :destroy` option. When we destroy an `Article`, we want its corresponding entries in `articles_categories` table to be deleted too.

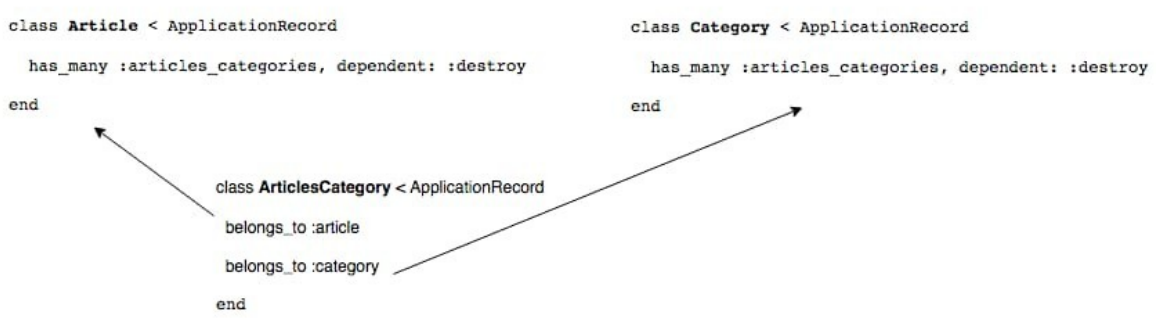
`has_many` in Category

The `Category` needs to be enhanced with a `has_many` association too, corresponding to the `belongs_to :category` inside the `ArticleCategory`:

```
1  # File: app/models/category.rb
2  #
3  class Category < ApplicationRecord
4    has_many :articles_categories, dependent: :destroy
5    validates :name, presence: true, uniqueness: {case_sensitive: false}
6  end
```

(the above code snippet online)

After the above, this is what we have done:



Articles And Categories With The Active Record Associations

All the above gives us interesting methods on each one of the models. See the following rails console session:

1. Get the first article and save into variable `a`:

```
 a = Article.first
   Article Load (0.7ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles"\
."id" ASC LIMIT $1  [["LIMIT", 1]]
  => #<Article id: 44, title: "Ruby", text: "Ruby for happiness", created_at: "20\
17-07-03 17:12:33", updated_at: "2017-07-03 17:12:33", user_id: 10>
```

1. Count the `articles_categories` of this Article:

```
 a.articles_categories.count
   (0.7ms)  SELECT COUNT(*) FROM "articles_categories" WHERE "articles_categories\
"."article_id" = $1  [["article_id", 44]]
 => 0
```

1. Create a programming Category:

```
  programming = Category.create! name: 'Programming'
     (0.2ms)  BEGIN
    Category Exists (1.7ms)  SELECT  1 AS one FROM "categories" WHERE LOWER("cate\
gories"."name") = LOWER($1) LIMIT $2  [["name", "Programming"], ["LIMIT", 1]]
    SQL (1.3ms)  INSERT INTO "categories" ("name", "created_at", "updated_at") VA\
LUES ($1, $2, $3) RETURNING "id"  [["name", "Programming"], ["created_at", "2017-\
07-03 19:48:35.145779"], ["updated_at", "2017-07-03 19:48:35.145779"]]
     (0.7ms)  COMMIT
  => #<Category id: 1, name: "Programming", created_at: "2017-07-03 19:48:35", up\
dated_at: "2017-07-03 19:48:35">
```

1. Add the programming Category to the Article `a`:

```
 a.articles_categories << ArticlesCategory.new(category: programming)
    (0.2ms)  BEGIN
   SQL (2.5ms)  INSERT INTO "articles_categories" ("article_id", "category_id", "\
created_at", "updated_at") VALUES ($1, $2, $3, $4) RETURNING "id"  [["article_id"\
, 44], ["category_id", 1], ["created_at", "2017-07-03 19:50:15.000825"], ["update\
d_at", "2017-07-03 19:50:15.000825"]]
    (0.3ms)  COMMIT
   ArticlesCategory Load (0.2ms)  SELECT  "articles_categories".* FROM "articles_\
categories" WHERE "articles_categories"."article_id" = $1 LIMIT $2  [["article_id\
", 44], ["LIMIT", 11]]
  => #<ActiveRecord::Associations::CollectionProxy [#<ArticlesCategory id: 1, art\
icle_id: 44, category_id: 1, created_at: "2017-07-03 19:50:15", updated_at: "2017\
-07-03 19:50:15">]>
```

See how we call `a.articles_categories` and we push a new item of type `ArticleCategory` which is saved in the database. 1. Count the `articles_categories` of the article again:

```
 a.articles_categories.count
    (1.0ms)  SELECT COUNT(*) FROM "articles_categories" WHERE "articles_categorie\
s"."article_id" = $1  [["article_id", 44]]
 => 1
```

The above proves the fact that we have attached the `ArticleCategory` to the Article `a`.

So, how do we get access to all categories of the Article `a`?

```
a.articles_categories.map {|ac| ac.category}
  ArticlesCategory Load (0.4ms)  SELECT "articles_categories".* FROM "articles_ca\
tegories" WHERE "articles_categories"."article_id" = $1  [["article_id", 44]]
 => [#<Category id: 1, name: "Programming", created_at: "2017-07-03 19:48:35", up\
dated_at: "2017-07-03 19:48:35">]
```

(the above code snippet online)

We had to go through the `articles_categories` in order to get access to the Categories of the Article at hand. This is what `:through` can help us with.

This is the new version of the `Article` class:

```
1  # File: app/models/article.rb
2  #
3  class Article < ApplicationRecord
4    belongs_to :user
5    has_many :articles_categories, dependent: :destroy
6    has_many :categories, through: :articles_categories
7
8    validates :user, presence: true
9    validates :title, presence: true
10   validates :text, presence: true
11 end
```

(the above code snippet online)

See line 6: `has_many :categories, through: :articles_categories`. This now allows us to do:

```
2.3.3 :002 > a.categories
  Category Load (1.7ms)  SELECT  "categories".* FROM "categories" INNER JOIN "art\
icles_categories" ON "categories"."id" = "articles_categories"."category_id" WHER\
E "articles_categories"."article_id" = $1 LIMIT $2  [["article_id", 44], ["LIMIT"\
, 11]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Category id: 1, name: "Progr\
amming", created_at: "2017-07-03 19:48:35", updated_at: "2017-07-03 19:48:35">]>
```

(the above code snippet online) i.e. `a.categories` and have access to the list of Categories of the particular Article without going through the `articles_categories`.

Closing on Associations

There are also some more advanced topics on model associations. Read about them in Rails Guides here.

Active Record Query Interface

We will close this chapter by expanding a little bit more on the programming interface that Active Record offers, in order to fetch data from the database.

Find By Id

```
2.3.3 :004 > article = Article.find(45)
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id\
" = $1 LIMIT $2  [["id", 45], ["LIMIT", 1]]
 => #<Article id: 45, title: "Ruby", text: "Ruby Arrays", created_at: "2017-07-03\
 20:01:49", updated_at: "2017-07-03 20:01:49", user_id: 10>
2.3.3 :005 >
```

(the above code snippet online)

The `Article.find(<integer>)` will try to locate the `Article` that has primary key equal to the number given as argument.

Note that if the record is not found, an `ActiveRecord::RecordNotFound` exception is going to be raised. See this:

```
2.3.3 :001 > Article.find(10)
  Article Load (0.6ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id\
" = $1 LIMIT $2  [["id", 10], ["LIMIT", 1]]
ActiveRecord::RecordNotFound: Couldn't find Article with 'id'=10
        from (irb):1
2.3.3 :002 >
```

(the above code snippet online)

In my case, there isn't any `Article` with id 10. That's why the `Article.find(10)` raises the `ActiveRecord::RecordNotFound` exception.

If you want to search for an `Article` by id, without actually this raising an exception if the id is not present, then you can use the method `.find_by_id` instead:

```
2.3.3 :005 > Article.find_by_id(10)
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id\
" = $1 LIMIT $2  [["id", 10], ["LIMIT", 1]]
 => nil
2.3.3 :006 >
```

(the above code snippet online)

`.first`

This method returns the first matching record. The ordering is by primary key. Note that the `.first` can take an integer as an argument. It will return that number of records from the database.

```
2.3.3 :001 > User.first(3)
  User Load (0.5ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
IT $1  [["LIMIT", 3]]
 => [#<User id: 10, email: "mary@gmail.com", password_digest: "$2a$10$YLIJQ0AEVCT\
FnDgYs013j.J32Hk9df2YPuvouECx.Tj...",
       created_at: "2017-06-10 06:46:05", updated_at: "2017-07-02 09:52:46", emai\
l_confirmation_token: "yStnErkVCFC9mYNHKFNYdA", terms_of_service: true>,
     #<User id: 12, email: "john@gmail.com", password_digest: "$2a$10$.JIoJKajyd2\
abxCkYFE0O.xaSyPLQ95ylXbOlyMDvdl...", created_at: "2017-06-10 08:02:26",
       updated_at: "2017-07-02 09:52:46", email_confirmation_token: nil, terms_of\
_service: true>,
     #<User id: 16, email: "paul@gmail.com",
       password_digest: "$2a$10$Ba9KZ9pGgiFuc5o8U8TDX.vv6SbLACUVrS89k7Yo8/d...", \
created_at: "2017-06-11 06:36:41", updated_at: "2017-07-02 09:52:46",
       email_confirmation_token: "0IRMYX9gcpVOtmn28kGLYw", terms_of_service: true\
>]
```

(the above code snippet online)

.last

Returns the last or last records from the database table.

.find_by

The `.find_by` locates the first record that matches the criteria given as input argument. For example:

```
2.3.3 :002 > User.find_by(email: 'john@gmail.com')
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."email" = $1 LI\
MIT $2  [["email", "john@gmail.com"], ["LIMIT", 1]]
 => #<User id: 12, email: "john@gmail.com", password_digest: "$2a$10$.JIoJKajyd2a\
bxCkYFE0O.xaSyPLQ95ylXbOlyMDvdl...", created_at: "2017-06-10 08:02:26",
      updated_at: "2017-07-02 09:52:46", email_confirmation_token: nil, terms_of_\
service: true>
```

(the above code snippet online)

The above call located the first user having email equal to `john@gmail.com`.

The next call....

```
2.3.3 :003 > User.find_by(email: 'john@gmail.com', terms_of_service: false)
  User Load (0.3ms)  SELECT  "users".* FROM "users" WHERE "users"."email" = $1 AN\
D "users"."terms_of_service" = $2 LIMIT $3  [["email", "john@gmail.com"], ["terms\
_of_service", "f"], ["LIMIT", 1]]
 => nil
```

(the above code snippet online)

tries to locate the user that has email `john@gmail.com` and `terms_of_service` equal to `false`.

The hash to `find_by` can contain any number of key/value pairs. All of them have to match in order for a record to be returned.

Fetching Multiple Objects

Iterating over all the records of a table might be easily written as follows:

```
User.all.each do |user|
 ... # process user
end
```

(the above code snippet online)

This piece of code is good when you know that you have few records, but does not scale. This is because it fetches all the records into the program memory. This usually ends in running out of memory and the Ruby program is usually killed by the operating system.

In order to process all the records of a table, try the `.find_each` method.

```
User.find_each do |user|
 .... # process user
end
```

(the above code snippet online)

This may look similar, but it is quite different in its implementation. It fetches the records in batches of 1000. Works out each batch and then fetches the next. Hence, it does not consume the memory of the program and can scale up to millions of records.

You can also specify the `batch_size` if you want:

```
User.find_each(batch_size: 100) do |user|
 .... # process user
end
```

(the above code snippet online)

Note that the order in which records are processed is in ascending primary key order.

Conditions

It is the `.where` method that is used to specify conditions for the records that we want to fetch. The conditions can be specified

1. as pure Strings
2. as an Array
3. as a Hash

Pure Strings

```
2.3.3 :022 >   User.where("email = 'john@gmail.com' and terms_of_service = true")\
.first
  User Load (0.6ms)  SELECT  "users".* FROM "users" WHERE (email = 'john@gmail.co\
m' and terms_of_service = true) ORDER BY "users"."id" ASC LIMIT $1  [["LIMIT", 1]\
]
 => #<User id: 12, email: "john@gmail.com", password_digest: "$2a$10$.JIoJKajyd2a\
bxCkYFE0O.xaSyPLQ95ylXbOlyMDvdl...",
       created_at: "2017-06-10 08:02:26", updated_at: "2017-07-02 09:52:46", emai\
l_confirmation_token: nil, terms_of_service: true>
```

(the above code snippet online)

The string `"email = 'john@gmail.com' and terms_of_service = true"` is the condition set we pass to `.where`. So, we want to bring all the records that match this condition set.

> **Important**: Do not build `.where` clauses using string interpolation. E.g. `.where("email = #{params[:email]}")` is bad. It allows hackers to take advantage of SQL injection and execute arbitrary SQL statements in your database.

Conditions As Arrays

```
User.where("email = ? and terms_of_service = ?", params[:email], true).first
```

(the above code snippet online)

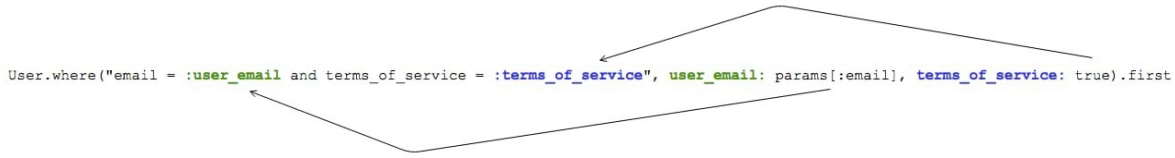
You can see that we give the values to substitute in the condition string as an array of values, after the condition string. The `?` in the condition string will be replaced by the values given in the array. Also, the values will be sanitized and escaped, which means there is no way that any security-breaching value can make it to the final SQL statement. Hence the problem of SQL injection goes away.

Also, if you want to make your Ruby statement more clear, you can use named placeholders in your string like this:

```
User.where("email = :user_email and terms_of_service = :terms_of_service", user_e\
mail: params[:email], terms_of_service: true).first
```

(the above code snippet online)

Do you see the placeholders and how they take their values?



Condition String With Placeholders

Conditions as Hash

You can pass a Hash with the key/value pairs that should be part of your condition. For example:

```
User.where(email: params[:email], terms_of_service: true).first
```

(the above code snippet online)

This is the equivalent to the previous example, but we are using a `Hash`, instead of a condition string and a Hash.

This is another example that uses a range as a value of your condition. It is translated to a `BETWEEN` operator in your database:

```
User.where(created_at: (Time.zone.now - 1.day)..Time.zone.now).first
```

(the above code snippet online)

The above will locate all the `User` records that their `created_at` attribute is within the range specified by `(Time.zone.now - 1.day)..Time.zone.now`. You can see below the generated SQL:

```
2.3.3 :026 >   User.where(created_at: (Time.zone.now - 1.day)..Time.zone.now).fir\
st
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE ("users"."created_at" B\
ETWEEN $1 AND $2) ORDER BY "users"."id" ASC LIMIT $3  [["created_at", "2017-07-03\
 19:48:13.381100"], ["created_at", "2017-07-04 19:48:13.396957"], ["LIMIT", 1]]
 => nil
```

(the above code snippet online)

Except from the range, which is translated to `BETWEEN`, you can use an array, which is translated to the `IN` operator:

```
2.3.3 :028 > User.where(email: ['john@gmail.com', 'mary@gmail.com'])
  User Load (0.7ms)  SELECT  "users".* FROM "users" WHERE "users"."email" IN ('jo\
hn@gmail.com', 'mary@gmail.com') LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: 10, email: "mary@gmail.com", password_di\
gest: "$2a$10$YLIJQ0AEVCTFnDgYs013j.J32Hk9df2YPuvouECx.Tj...", created_at: "2017-\
06-10 06:46:05", updated_at: "2017-07-02 09:52:46", email_confirmation_token: "yS\
tnErkVCFC9mYNHKFNYdA", terms_of_service: true>, #<User id: 12, email: "john@gmail\
.com", password_digest: "$2a$10$.JIoJKajyd2abxCkYFE0O.xaSyPLQ95ylXbOlyMDvdl...", \
created_at: "2017-06-10 08:02:26", updated_at: "2017-07-02 09:52:46", email_confi\
rmation_token: nil, terms_of_service: true>]>
```

(the above code snippet online)

The condition `where(email: ['john@gmail.com', 'mary@gmail.com'])` has been translated to `where users.email in ('john@gmail.com', 'mary@gmail.com')`.

Not Condition

You can specify the negation of a condition using the `.not` method:

```
2.3.3 :029 > User.where.not(email: ['john@gmail.com', 'mary@gmail.com'])
  User Load (0.5ms)  SELECT  "users".* FROM "users" WHERE ("users"."email" NOT IN\
 ('john@gmail.com', 'mary@gmail.com')) LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: 16, email: "paul@gmail.com", password_di\
gest: "$2a$10$Ba9KZ9pGgiFuc5o8U8TDX.vv6SbLACUVrS89k7Yo8/d...",
    created_at: "2017-06-11 06:36:41", updated_at: "2017-07-02 09:52:46", email_c\
onfirmation_token: "0IRMYX9gcpVOtmn28kGLYw", terms_of_service: true>,
    ...
```

(the above code snippet online)

The `.where.not(email: ['john@gmail.com', 'mary@gmail.com'])` has been translated to `WHERE ("users"."email" NOT IN ('john@gmail.com', 'mary@gmail.com'))`.

Ordering

The `.order` method is used to order the results before they are returned to the Ruby world. Corresponds to the `ORDER BY` clause in the database.

```
2.3.3 :001 > Article.order(:title)
  Article Load (0.5ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
"title" ASC LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<Article id: 45, title: "Ruby", text: "Ruby Arrays\
", created_at: "2017-07-03 20:01:49", updated_at: "2017-07-03 20:01:49", user_id:\
 10>]>
```

(the above code snippet online)

As you can see above, the `.order(:title)` is translated to `ORDER BY "articles"."title" ASC`.

If we want to order in descending order, we do it as follows:

```
2.3.3 :002 > Article.order(title: :desc)
  Article Load (0.7ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
"title" DESC LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<Article id: 45, title: "Ruby", text: "Ruby Arrays\
", created_at: "2017-07-03 20:01:49", updated_at: "2017-07-03 20:01:49", user_id:\
 10>]>
```

(the above code snippet online)

The `.order(title: :desc)` has been translated to `ORDER BY "articles"."title" DESC`.

Selecting Fields From Result Set

Sometimes a record in the database may have many columns with long values. It may be inefficient to be bringing all the fields, if you don't want all of them, for each one of the records. Most of the time we are interested in specific fields of these records.

The `.select` method allows you to specify which fields to fetch:

```
2.3.3 :005 >   Article.order(title: :desc).select("id, title")
  Article Load (0.5ms)  SELECT  id, title FROM "articles" ORDER BY "articles"."ti\
tle" DESC LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<Article id: 45, title: "Ruby">]>
```

(the above code snippet online)

As you can see, the `.select("id, title")` has been translated to `SELECT id, title` in the SQL query issued to the database.

However, please, note that when you use `select`, the columns that you don't bring in, are not accessible and Rails raises the exception `ActiveModel::MissingAttributeError` if you try to access them:

```
2.3.3 :006 > a = Article.order(title: :desc).select("id, title").first
  Article Load (0.5ms)  SELECT  id, title FROM "articles" ORDER BY "articles"."ti\
tle" DESC LIMIT $1  [["LIMIT", 1]]
 => #<Article id: 45, title: "Ruby">
2.3.3 :007 > a.id
 => 45
2.3.3 :008 > a.title
 => "Ruby"
2.3.3 :009 > a.text
ActiveModel::MissingAttributeError: missing attribute: text
        from (irb):9
```

(the above code snippet online)

Do you see how the exception `ActiveModel::MissingAttributeError` is raised because of the call to `a.text`?

Limit and Offset

Active Record offers you the `limit` and `offset` methods which correspond to the `limit` and `offset` methods in your underlying database management system.

Try this one:

```
2.3.3 :012 > User.order(:email).offset(5).limit(1)
  User Load (0.5ms)  SELECT  "users".* FROM "users" ORDER BY "users"."email" ASC \
LIMIT $1 OFFSET $2  [["LIMIT", 1], ["OFFSET", 5]]
 => #<ActiveRecord::Relation [#<User id: 17, email: "kristin@gmail.com", password\
_digest: "$2a$10$COrxjcEMPOh/t3u.Hf5uhuW.NaQ4C79EzGB.s0rGHFT...", created_at: "20\
17-06-11 06:55:22", updated_at: "2017-07-02 09:52:46", email_confirmation_token: \
"nVBfAifMQBaRh86B9dzWMg", terms_of_service: true>]>
```

(the above code snippet online)

The `.order(:email).offset(5).limit(1)` is translated to `ORDER BY "users"."email" ASC LIMIT $1 OFFSET $2 [["LIMIT", 1], ["OFFSET", 5]]`, which basically brings the 6th User (in ordering terms of email in ascending order).

The `limit` and `offset` can be used to implement pagination.

Group

The `.group` method can be used to generate SQL statements that incorporate `GROUP BY` clauses.

```
2.3.3 :022 > User.select("date(created_at) as creation_date, count(id) as number_\
of_users").group("date(created_at)")
  User Load (0.5ms)  SELECT  date(created_at) as creation_date, count(id) as numb\
er_of_users FROM "users" GROUP BY date(created_at) LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: nil>, #<User id: nil>, #<User id: nil>, \
#<User id: nil>]>
```

(the above code snippet online)

The `.group("date(created_at)")` has been translated to `GROUP BY date(created_at)`.

Note that `.group` call is usually used with `.select` because with `.select` we can tell which column to return, something that is related to the way we group our results.

Having

The `.having` is used to apply conditions to statements with `.group`. It takes arguments like the `.where` clause does.

Locking Records for Update

Locking is very important when two users try to update the same resource. So it prevents race conditions and makes sure that data are kept consistent.

Rails offers two mechanisms for locking.

1. Optimistic Locking
2. Pessimistic Locking

Optimistic Locking

In optimistic locking we allow multiple users to start editing the same resource, but we give the update privilege to only one of these users.

Let's implement optimistic locking for the `Article` resource.

Add `lock_version` Column

In order to use optimistic locking for the `Article` resource, we need to add the column `lock_version`.

```
blog $ bin/rails generate migration add_lock_version_to_articles
      invoke  active_record
      create    db/migrate/20170705051250_add_lock_version_to_articles.rb
blog $
```

(the above code snippet online)

We now edit the file `db/migrate/20170705051250_add_lock_version_to_articles.rb` to add the `lock_version` column:

```ruby
# File: db/migrate/20170705051250_add_lock_version_to_articles.rb
#
class AddLockVersionToArticles < ActiveRecord::Migration[5.1]
  def change
    add_column :articles, :lock_version, :bigint, default: 1, null: false
  end
end
```

(the above code snippet online)

We use the `add_column` method to add a big integer column named `lock_version`. Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170705051250 AddLockVersionToArticles: migrating ========================
-- add_column(:articles, :lock_version, :bigint, {:default=>1, :null=>false})
   -> 0.0238s
== 20170705051250 AddLockVersionToArticles: migrated (0.0239s) ===============

blog $
```

(the above code snippet online)

And then let's confirm the column in the table:

```
blog $ psql -d blog_development -c "\d+ articles"
                                                          Table "public.articles"
    Column     |            Type             |                      Modifiers    \
                   | Storage  | Stats target | Description
--------------+----------------------------+-----------------------------------\
-------------------+----------+--------------+-------------
 id           | bigint                      | not null default nextval('articles_\
id_seq'::regclass) | plain    |              |
 title        | character varying           | not null                          \
                   | extended |              |
 text         | character varying           | not null                          \
                   | extended |              |
 created_at   | timestamp without time zone | not null                          \
                   | plain    |              |
 updated_at   | timestamp without time zone | not null                          \
                   | plain    |              |
 user_id      | bigint                      |                                   \
                   | plain    |              |
 lock_version | bigint                      | not null default 1                \
                   | plain    |              |
Indexes:
    "articles_pkey" PRIMARY KEY, btree (id)
    "articles_user_id_idx" btree (user_id)
Foreign-key constraints:
    "articles_user_id_users_id_fk" FOREIGN KEY (user_id) REFERENCES users(id)
Referenced by:
    TABLE "articles_categories" CONSTRAINT "articles_categories_article_fk" FOREI\
GN KEY (article_id) REFERENCES articles(id)

blog $
```

(the above code snippet online)

You can see the last column in the list, `lock_version` of type `bigint`.

With the `lock_version` in place, let's run a `bin/rails console` session that would prove the optimistic locking works:

```
blog $ bin/rails console
2.3.3 :001 > article1_ref_1 = Article.first
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
"id" ASC LIMIT $1  [["LIMIT", 1]]
 => #<Article id: 45, title: "Ruby", text: "Ruby Arrays", created_at: "2017-07-03\
 20:01:49", updated_at: "2017-07-03 20:01:49", user_id: 10, lock_version: 1>
2.3.3 :002 > article1_ref_2 = Article.find(45)
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id\
" = $1 LIMIT $2  [["id", 45], ["LIMIT", 1]]
 => #<Article id: 45, title: "Ruby", text: "Ruby Arrays", created_at: "2017-07-03\
 20:01:49", updated_at: "2017-07-03 20:01:49", user_id: 10, lock_version: 1>
2.3.3 :003 > article1_ref_1.title = 'Ruby Arrays'
 => "Ruby Arrays"
2.3.3 :004 > article1_ref_1.save
   (0.2ms)  BEGIN
  User Load (0.5ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = $1 LIMIT\
 $2  [["id", 10], ["LIMIT", 1]]
  SQL (0.6ms)  UPDATE "articles" SET "title" = 'Ruby Arrays', "updated_at" = '201\
7-07-05 05:21:47.160952', "lock_version" = 2 WHERE "articles"."id" = $1 AND "arti\
cles"."lock_version" = $2  [["id", 45], ["lock_version", 1]]
   (1.1ms)  COMMIT
 => true
2.3.3 :005 > article1_ref_2.title = 'Ruby Programming'
 => "Ruby Programming"
2.3.3 :006 > article1_ref_2.save
   (0.2ms)  BEGIN
  User Load (0.6ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = $1 LIMIT\
 $2  [["id", 10], ["LIMIT", 1]]
  SQL (0.4ms)  UPDATE "articles" SET "title" = 'Ruby Programming', "updated_at" =\
 '2017-07-05 05:22:23.110502', "lock_version" = 2 WHERE "articles"."id" = $1 AND \
"articles"."lock_version" = $2  [["id", 45], ["lock_version", 1]]
   (0.1ms)  ROLLBACK
ActiveRecord::StaleObjectError: Attempted to update a stale object: Article.
        from (irb):7
```

(the above code snippet online)

We use 2 references to access the same article. `article1_ref_1` and `article1_ref_2`. Both references hold the first `Article` and they assume that its `lock_version` column is `1`.

We then use the first reference to update the `title` of the `Article`. You can see now that the update that is executed in the database makes sure that the record being updated has the `lock_version` equal to `1`. See command `004` the `WHERE` clause:

```
WHERE "articles"."id" = $1 AND "articles"."lock_version" = $2  [["id", 45], ["loc\
k_version", 1]]
```

(the above code snippet online)

Not only that, this first update, increases the `lock_version` to `2`.

When the second `#save` command is tried on command `006`, the `WHERE` clause assumes `lock_version` equal to `1`, because the reference `article1_ref_2` does not know that the previous update has taken place and it still has its `lock_version` to `1`. For that reason, the `UPDATE` fails and the exception `ActiveRecord::StaleObjectError` exception is raised.

Pessimistic Locking

The pessimistic locking does not enforce you to enhance your schema with any special column. It uses the underlying database locking mechanisms. You need to call the method `.lock` within a transaction block in order to lock the record that you want to update and prevent others from reading or updating it.

> **Important**: Let's rollback the previous migration that added the `lock_version` to the `Article` table. `bash blog $ bin/rake db:rollback == 20170705051250 AddLockVersionToArticles: reverting ========== -- remove_column(:articles, :lock_version, :bigint, {:default=>1, :null=>false}) -> 0.0012s == 20170705051250 AddLockVersionToArticles: reverted (0.0047s) ========== blog $ bin/rails destroy migration add_lock_version_to_articles invoke active_record remove db/migrate/20170705051250_add_lock_version_to_articles.rb`

Now, let's start a `bin/rails console` and try to use pessimistic locking for the update of the first `Article`:

```
2.3.3 :019 >   Article.transaction do
2.3.3 :020 >     article = Article.lock.first
2.3.3 :021?>     article.title = 'Ruby Programming'
2.3.3 :022?>     article.save!
2.3.3 :023?>   end
   (0.2ms)  BEGIN
  Article Load (0.5ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
"id" ASC LIMIT $1 FOR UPDATE  [["LIMIT", 1]]
  User Load (0.3ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = $1 LIMIT\
 $2  [["id", 10], ["LIMIT", 1]]
  SQL (0.3ms)  UPDATE "articles" SET "title" = $1, "updated_at" = $2 WHERE "artic\
les"."id" = $3  [["title", "Ruby Programming"], ["updated_at", "2017-07-05 05:40:\
00.023612"], ["id", 45]]
   (0.4ms)  COMMIT
 => true
```

(the above code snippet online)

The `Article.transaction` is a way for us to start a database transaction. All the SQL commands that would be generated by the Ruby block that is given to `Article.transaction` they would be executed as a block, atomically, all or nothing.

Note that we don't just fetch the first record with `Article.first`, but we lock it and fetch it: `Article.lock.first`. In this case, the `SELECT` generated has a `FOR UPDATE` at the end. This makes sure that the database locks the record and no other client application/code can read it and update it until the transaction is finished.

Note that the `.lock` method can take an argument which might be a database specific lock mode. For example, in MySQL, you can give the string `'LOCK IN SHARE MODE'`, which allows others to read but not update the record while the transaction is running. For PostgreSQL, the equivalent string is `'FOR SHARE'`

Joining Tables

We have already learned about `JOIN` and `LEFT JOIN` in our SQL chapter. How can we join information from different tables using Active Record? Let's see how.

Method `.joins`

The method `.joins` is very powerful.

For example, you can go with pure SQL string argument.

```
2.3.3 :037 > Article.joins("join users on users.id = articles.user_id")
  Article Load (0.8ms)  SELECT  "articles".* FROM "articles" join users on users.\
id = articles.user_id LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<Article id: 45, title: "Ruby Arrays", text: "Ruby\
 Arrays", created_at: "2017-07-03 20:01:49",
                                updated_at: "2017-07-05 05:50:44", user_id: 10>,
                               #<Article id: 50, title: "System 1 and System 2", t\
ext: "System 1 and System 2 modes of thinking",
                                created_at: "2017-07-05 17:08:50", updated_at: "2\
017-07-05 17:08:50", user_id: 18>,
                               #<Article id: 49, title: "Team Building", text: "Te\
am building events", created_at: "2017-07-05 17:08:21",
                                updated_at: "2017-07-05 17:08:21", user_id: 18>, \
#<Article id: 48, title: "Leadership Authenticity",
                                text: "Authenticity in Leadership", created_at: "\
2017-07-05 17:06:35", updated_at: "2017-07-05 17:06:35", user_id: 21>,
                               #<Article id: 47, title: "Unconscious Bias", text: \
"Unconscious Bias in Business Environment", created_at: "2017-07-05 17:05:50",
                                updated_at: "2017-07-05 17:05:50", user_id: 21>, \
#<Article id: 46, title: "Business Psychology",
                                text: "Psychology in Business", created_at: "2017\
-07-05 17:05:14", updated_at: "2017-07-05 17:05:30", user_id: 21>]>
```

(the above code snippet online)

We have called the `.joins` method on the `Article` class and we have given a *join* SQL string as argument. The SQL *join* string was `"join users on users.id = articles.user_id"`. This is what has been executed in the database:

```
SELECT  "articles".* FROM "articles" join users on users.id = articles.user_id LI\
MIT $1  [["LIMIT", 11]]
```

(the above code snippet online)

which brings the list of articles that have been written by a user.

This is another example:

```
2.3.3 :039 > User.joins("join articles on articles.user_id = users.id")
  User Load (1.2ms)  SELECT  "users".* FROM "users" join articles on articles.use\
r_id = users.id LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: 10, email: "mary@gmail.com", password_di\
gest: "$2a$10$YLIJQ0AEVCTFnDgYs013j.J32Hk9df2YPuvouECx.Tj...",
                              created_at: "2017-06-10 06:46:05", updated_at: "201\
7-07-02 09:52:46", email_confirmation_token: "yStnErkVCFC9mYNHKFNYdA",
                              terms_of_service: true>,
                              #<User id: 18, email: "angela@gmail.com", password_\
digest: "$2a$10$HXoMZTFWqPqxobuUFI7yRuNd83Q.xoyvm1MXSHT6PBw...",
                              created_at: "2017-06-11 06:58:18", updated_at: "201\
7-07-02 09:52:46", email_confirmation_token: nil, terms_of_service: true>,
                              #<User id: 18, email: "angela@gmail.com", password_\
digest: "$2a$10$HXoMZTFWqPqxobuUFI7yRuNd83Q.xoyvm1MXSHT6PBw...",
                              created_at: "2017-06-11 06:58:18", updated_at: "201\
7-07-02 09:52:46", email_confirmation_token: nil, terms_of_service: true>,
                              #<User id: 21, email: "kelly@gmail.com", password_d\
igest: "$2a$10$7ajlqQuSh/ZkMuVulje9B.deuNVX0EGZsDSm0kWFx8D...",
                              created_at: "2017-06-11 09:45:22", updated_at: "201\
7-07-02 09:52:46", email_confirmation_token: nil, terms_of_service: true>,
                              #<User id: 21, email: "kelly@gmail.com", password_d\
igest: "$2a$10$7ajlqQuSh/ZkMuVulje9B.deuNVX0EGZsDSm0kWFx8D...",
                              created_at: "2017-06-11 09:45:22", updated_at: "201\
7-07-02 09:52:46", email_confirmation_token: nil, terms_of_service: true>,
                              #<User id: 21, email: "kelly@gmail.com", password_d\
igest: "$2a$10$7ajlqQuSh/ZkMuVulje9B.deuNVX0EGZsDSm0kWFx8D...",
                              created_at: "2017-06-11 09:45:22", updated_at: "201\
7-07-02 09:52:46", email_confirmation_token: nil, terms_of_service: true>]>
```

(the above code snippet online)

This brings the users that have at least one article. Obviously, the problem with this query is that it brings the same user multiple times, if a user has multiple articles written.

You can also use the `.joins` method and, instead of specifying the `JOIN` using SQL, you can use the association symbols. Let's see that:

```
2.3.3 :041 > Article.joins(:user)
  Article Load (0.9ms)  SELECT  "articles".* FROM "articles" INNER JOIN "users" O\
N "users"."id" = "articles"."user_id" LIMIT $1  [["LIMIT", 11]]
```

(the above code snippet online)

So, the `Article.joins(:user)` is equal to `Article.joins("join users on users.id = articles.id")`.

Similarly, try the next one:

```
2.3.3 :044 >   User.joins(:articles)
      User Load (0.8ms)  SELECT  "users".* FROM "users" INNER JOIN "articles" ON \
"articles"."user_id" = "users"."id" LIMIT $1  [["LIMIT", 11]]
```

(the above code snippet online)

You can join the first model to many other tables, if this is supported by your design. For example `Article.joins(:categories, :comments)` would join `Article` both with `Categories` and `Comments`, if you had the `Comment` model in your design.

Also, you can carry out chain joins. For example:

```
2.3.3 :051 > User.joins(articles: :articles_categories)
  User Load (0.9ms)  SELECT  "users".* FROM "users" INNER JOIN "articles" ON "art\
icles"."user_id" = "users"."id" INNER JOIN "articles_categories" ON "articles_cat\
egories"."article_id" = "articles"."id" LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation []>
```

(the above code snippet online)

The `User.joins(articles: :articles_categories)` joins users with `articles` and then `articles` with `articles_categories`.

And you can expand like this:

```
2.3.3 :053 > User.joins(articles: {articles_categories: :category})
  User Load (1.9ms)  SELECT  "users".* FROM "users" INNER JOIN "articles" ON "art\
icles"."user_id" = "users"."id" INNER JOIN "articles_categories" ON "articles_cat\
egories"."article_id" = "articles"."id" INNER JOIN "categories" ON "categories"."\
id" = "articles_categories"."category_id" LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation []>
```

(the above code snippet online)

The `User.joins(articles: {articles_categories: :category})` joins `users` to `articles`, `articles` to `articles_categories` and `articles_categories` to `categories`.

But, on the above case, and since you have `:through` implemented in your `Article` model, the call could be simplified as:

```
2.3.3 :054 > User.joins(articles: :categories)
  User Load (2.0ms)  SELECT  "users".* FROM "users" INNER JOIN "articles" ON "art\
icles"."user_id" = "users"."id"
   INNER JOIN "articles_categories" ON "articles_categories"."article_id" = "arti\
cles"."id"
   INNER JOIN "categories" ON "categories"."id" = "articles_categories"."category\
_id" LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation []>
```

(the above code snippet online)

As you can see, the `User.joins(articles: :categories)` generates the exact same SQL like the one generated from `User.joins(articles: {articles_categories: :category})`.

Chaining `.where`

Of course, you can chain `.where` clause in `.joins` calls in order to limit the result set returned.

For example:

```
2.3.3 :077 > User.joins(articles: :categories).where("categories.name = :category\
_name", category_name: 'Programming')
  User Load (1.2ms)  SELECT  "users".* FROM "users"
                     INNER JOIN "articles" ON "articles"."user_id" = "users"."id"
                     INNER JOIN "articles_categories" ON "articles_categories"."a\
rticle_id" = "articles"."id"
                     INNER JOIN "categories" ON "categories"."id" = "articles_cat\
egories"."category_id"
                     WHERE (categories.name = 'Programming') LIMIT $1  [["LIMIT",\
 11]]
 => #<ActiveRecord::Relation [#<User id: 21, email: "kelly@gmail.com", password_d\
igest: "$2a$10$7ajlqQuSh/ZkMuVulje9B.deuNVX0EGZsDSm0kWFx8D...",
 created_at: "2017-06-11 09:45:22", updated_at: "2017-07-02 09:52:46", email_conf\
irmation_token: nil, terms_of_service: true>]>
```

(the above code snippet online)

Do you see the `.where("categories.name = :category_name", category_name: 'Programming')` ?

This could have been written also like this:

```
2.3.3 :082 > User.joins(articles: :categories).where(categories: {name: 'Programm\
ing'})
  User Load (1.1ms)  SELECT  "users".* FROM "users"
                     INNER JOIN "articles" ON "articles"."user_id" = "users"."id"
                     INNER JOIN "articles_categories" ON "articles_categories"."a\
rticle_id" = "articles"."id"
                     INNER JOIN "categories" ON "categories"."id" = "articles_cat\
egories"."category_id"
                     WHERE "categories"."name" = $1 LIMIT $2  [["name", "Programm\
ing"], ["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: 21, email: "kelly@gmail.com", password_d\
igest: "$2a$10$7ajlqQuSh/ZkMuVulje9B.deuNVX0EGZsDSm0kWFx8D...", created_at: "2017\
-06-11 09:45:22", updated_at: "2017-07-02 09:52:46", email_confirmation_token: ni\
l, terms_of_service: true>]>
```

(the above code snippet online)

In the above example, instead of using SQL string for the `.where` clause, we specify the condition using a `Hash`.

`.left_outer_joins`

The method allows you to carry out `LEFT OUTER JOIN` joins. For example:

```
2.3.3 :087 > User.left_outer_joins(:articles)
  User Load (3.0ms)  SELECT  "users".* FROM "users" LEFT OUTER JOIN "articles" ON\
 "articles"."user_id" = "users"."id" LIMIT $1  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: 10, email: "mary@gmail.com", password_di\
gest: "$2a$10$YLIJQ0AEVCTFnDgYs013j.J32Hk9df2YPuvouECx.Tj...",
      created_at: "2017-06-10 06:46:0...
```

(the above code snippet online)

The above `.left_outer_joins` generates a `LEFT OUTER JOIN` join and returns all the users alongside their articles, including the users that they do not have any article.

Eager Loading Associated Objects

Let's try the following code:

```
User.all.each do |user|
  puts user.articles.first.try(:title)
end
```

(the above code snippet online)

This might look good at first read, but it has a drawback. It suffers from the N + 1 queries problem.

Let's execute it and you will see what I mean:

```
2.3.3 :094 > User.all.each do |user|
2.3.3 :095 >     puts user.articles.first.try(:title)
2.3.3 :096?>   end
  User Load (0.8ms)  SELECT "users".* FROM "users"
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 10], ["LIMIT", 1]\
]
Ruby Arrays
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 12], ["LIMIT", 1]\
]

  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 16], ["LIMIT", 1]\
]

  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 17], ["LIMIT", 1]\
]

  Article Load (0.3ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 18], ["LIMIT", 1]\
]
Team Building
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 19], ["LIMIT", 1]\
]

  Article Load (0.2ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 20], ["LIMIT", 1]\
]

  Article Load (0.2ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 21], ["LIMIT", 1]\
]
Business Psychology
  Article Load (0.2ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 24], ["LIMIT", 1]\
]

  Article Load (0.3ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."us\
er_id" = $1 ORDER BY "articles"."id" ASC LIMIT $2  [["user_id", 26], ["LIMIT", 1]\
]
```

(the above code snippet online)

It executes 1 query to bring all the users, and then 10 queries, for each one of the 10 users, to

bring users articles.

If we know that we will need the associated object, it is better if we write the Ruby code so that the underlying queries are the minimum required.

This is how we should write this query in order to avoid this *N + 1* queries problem:

```
2.3.3 :097 > User.includes(:articles).all.each do |user|
2.3.3 :098 >     puts user.articles.first.try(:title)
2.3.3 :099?>   end
  User Load (0.6ms)  SELECT "users".* FROM "users"
  Article Load (1.1ms)  SELECT "articles".* FROM "articles" WHERE "articles"."use\
r_id" IN (10, 12, 16, 17, 18, 19, 20, 21, 24, 26)
```

(the above code snippet online)

So, we call `User.includes(:articles)` instead of just browsing the users and then loading the articles of each user. As you can see, the queries are now only 2. The second query brings all the articles of the users at hand.

The `.includes()` can take similar arguments like the `.joins()`. Also, it can be chained with `.where()` clauses.

Scopes

Scopes are named queries that you can define at the class level of a Model. For example:

```
1  # File: app/models/article.rb
2  #
3  class Article < ApplicationRecord
4    belongs_to :user
5    has_many :articles_categories, dependent: :destroy
6    has_many :categories, through: :articles_categories
7    scope :programming, -> { joins(:categories).where(categories: {name: 'Programmi\
8  ng'}) }
9
10   validates :user, presence: true
11   validates :title, presence: true
12   validates :text, presence: true
13 end
```

(the above code snippet online)

Look at line 7. It defines the *scope* with name `:programming`. We give the definition of the query inside a lambda block. Then we can use the class method `.programming` to get the ActiveRecord Relation corresponding to this particular query:

```
2.3.3 :001 > Article.programming
  Article Load (1.0ms)  SELECT  "articles".* FROM "articles"
      INNER JOIN "articles_categories" ON "articles_categories"."article_id" = "a\
rticles"."id"
      INNER JOIN "categories" ON "categories"."id" = "articles_categories"."categ\
ory_id"
      WHERE "categories"."name" = $1 LIMIT $2  [["name", "Programming"], ["LIMIT"\
, 11]]
 => #<ActiveRecord::Relation [#<Article id: 47, title: "Unconscious Bias", text: \
"Unconscious Bias in Business Environment", created_at: "2017-07-05 17:05:50", up\
dated_at: "2017-07-05 17:05:50", user_id: 21>]>
2.3.3 :002 > Article.programming.count
   (1.2ms)  SELECT COUNT(*) FROM "articles"
      INNER JOIN "articles_categories" ON "articles_categories"."article_id" = "a\
rticles"."id"
      INNER JOIN "categories" ON "categories"."id" = "articles_categories"."categ\
ory_id"
      WHERE "categories"."name" = $1  [["name", "Programming"]]
 => 1
```

(the above code snippet online)

Scopes are basically equivalent to class methods. So, the previous one could have been equally written as:

```
def self.programming
  joins(:categories).where(categories: {name: 'Programming'})
end
```

(the above code snippet online)

So, it is up to your own taste and to the guidelines of your team which method to prefer.

Dynamic Finders

We have already seen this feature. Active Record supports dynamic finders like for example the following:

```
2.3.3 :005 > Article.find_by_title('Ruby Arrays')
  Article Load (0.4ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."ti\
tle" = $1 LIMIT $2  [["title", "Ruby Arrays"], ["LIMIT", 1]]
 => #<Article id: 45, title: "Ruby Arrays", text: "Ruby Arrays", created_at: "201\
7-07-03 20:01:49", updated_at: "2017-07-05 05:50:44", user_id: 10>
```

(the above code snippet online)

The `Article.find_by_title` method does not actually exist. Active Record uses the `method_missing` technique to treat this method dynamically. It identifies that the method starts with

`find_by` and parses the name of the method to identify the attributes we want to query with. So, the `.find_by_title` means that we want to locate Article by their title, whereas the method `.find_by_title_and_text` means that we want to locate Article by their title and text.

Enums

An enumeration allows to map an integer column to different possible values. Let's do that for the `Article` class after first having introduced the `state` column:

```
blog $ bin/rails generate migration add_state_to_articles
      invoke  active_record
      create    db/migrate/20170705194359_add_state_to_articles.rb
blog $
```

(the above code snippet online)

We edit the file `db/migrate/20170705194359_add_state_to_articles.rb` and we set the following content:

```ruby
# File: db/migrate/20170705194359_add_state_to_articles.rb
#
class AddStateToArticles < ActiveRecord::Migration[5.1]
  def change
    add_column :articles, :state, :integer, null: false, default: 0
  end
end
```

(the above code snippet online)

Now, let's run the migration:

```
blog $ bin/rake db:migrate
== 20170705194359 AddStateToArticles: migrating ================================
-- add_column(:articles, :state, :integer, {:null=>false, :default=>0})
   -> 0.0274s
== 20170705194359 AddStateToArticles: migrated (0.0276s) =======================

blog $
```

(the above code snippet online)

We now go to the `Article` model and we declare the `state` as an `enum`:

```
1  # File: app/models/article.rb
2  #
3  class Article < ApplicationRecord
4    belongs_to :user
5    has_many :articles_categories, dependent: :destroy
6    has_many :categories, through: :articles_categories
7    scope :programming, -> { joins(:categories).where(categories: {name: 'Programmi\
8  ng'}) }
9    enum state: [:draft, :published]
10
11   validates :user, presence: true
12   validates :title, presence: true
13   validates :text, presence: true
14 end
```

(the above code snippet online)

Look at line 8, the `enum state: %i[draft published]`. This tells Rails that the `state` attribute takes one of the values `:draft` or `:published`.

Let's see what extra facilities we get with this:

```
1  2.3.3 :001 > Article.draft
2    Article Load (0.5ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."st\
3  ate" = $1 LIMIT $2  [["state", 0], ["LIMIT", 11]]
4   => #<ActiveRecord::Relation [#<Article id: 46, title: "Business Psychology", tex\
5  t: "Psychology in Business", created_at: "2017-07-05 17:05:14", updated_at: "2017\
6  -07-05 17:05:30", user_id: 21, state: "draft">, #<Article id: 47, title: "Unconsc\
7  ious Bias", text: "Unconscious Bias in Business Environment", created_at: "2017-0\
8  7-05 17:05:50", updated_at: "2017-07-05 17:05:50", user_id: 21, state: "draft">, \
9  #<Article id: 48, title: "Leadership Authenticity", text: "Authenticity in Leader\
10 ship", created_at: "2017-07-05 17:06:35", updated_at: "2017-07-05 17:06:35", user\
11 _id: 21, state: "draft">, #<Article id: 49, title: "Team Building", text: "Team b\
12 uilding events", created_at: "2017-07-05 17:08:21", updated_at: "2017-07-05 17:08\
13 :21", user_id: 18, state: "draft">, #<Article id: 50, title: "System 1 and System\
14  2", text: "System 1 and System 2 modes of thinking", created_at: "2017-07-05 17:\
15 08:50", updated_at: "2017-07-05 17:08:50", user_id: 18, state: "draft">]>
16 2.3.3 :002 > Article.published
17   Article Load (0.6ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."st\
18 ate" = $1 LIMIT $2  [["state", 1], ["LIMIT", 11]]
19  => #<ActiveRecord::Relation [#<Article id: 45, title: "Ruby Arrays", text: "Ruby\
20  Arrays", created_at: "2017-07-03 20:01:49", updated_at: "2017-07-05 19:51:40", u\
21 ser_id: 10, state: "published">]>
22 2.3.3 :003 > article = Article.first
23   Article Load (0.4ms)  SELECT  "articles".* FROM "articles" ORDER BY "articles".\
24 "id" ASC LIMIT $1  [["LIMIT", 1]]
25  => #<Article id: 45, title: "Ruby Arrays", text: "Ruby Arrays", created_at: "201\
```

```
7-07-03 20:01:49", updated_at: "2017-07-05 19:51:40", user_id: 10, state: "publis\
hed">
2.3.3 :004 > article.published?
 => true
2.3.3 :005 > article.draft?
 => false
2.3.3 :006 > article.draft!
   (0.3ms)  BEGIN
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = $1 LIMIT\
 $2  [["id", 10], ["LIMIT", 1]]
  SQL (0.7ms)  UPDATE "articles" SET "updated_at" = $1, "state" = $2 WHERE "artic\
les"."id" = $3  [["updated_at", "2017-07-05 19:53:18.932727"], ["state", 0], ["id\
", 45]]
   (0.3ms)  COMMIT
 => true
2.3.3 :007 > article.published!
   (0.2ms)  BEGIN
  SQL (0.5ms)  UPDATE "articles" SET "updated_at" = $1, "state" = $2 WHERE "artic\
les"."id" = $3  [["updated_at", "2017-07-05 19:53:27.265501"], ["state", 1], ["id\
", 45]]
   (2.2ms)  COMMIT
 => true
2.3.3 :008 >
```

(the above code snippet online)

1. `Article.draft` brings the articles that have "draft" for their state, or actual value "0". The result is an ActiveRecord::Relation and can be chained with other ActiveRecord::Relation methods, like `.joins` or `.where`.
2. `Article.published` brings the articles that have "published" for their state, or actual value "1".
3. `article.published?` tells us whether the `article` is on state `published`.
4. `article.draft?` tells us whether the `article` is on state `draft`.
5. `article.draft!` will update the state of the article to be `draft`.
6. `article.published!` will update the state of the article to be `published`.

> *Information:* In order to keep track of the state like above, you may want to better add an index on the `state` column so that the queries run faster.

Find or Build/Create a New Object

Very often, you want to find a record and, if this is not found, to be created. This is called *find or create by*.

The methods that do this are two, the `find_or_create_by` and the `find_or_create_by!`.

Let's see an example with `find_or_create_by`:

```
2.3.3 :011 > Category.find_or_create_by(name: 'Leadership')
  Category Load (1.7ms)  SELECT  "categories".* FROM "categories" WHERE "categori\
es"."name" = $1 LIMIT $2  [["name", "Leadership"], ["LIMIT", 1]]
   (0.2ms)  BEGIN
  Category Exists (1.8ms)  SELECT  1 AS one FROM "categories" WHERE LOWER("catego\
ries"."name") = LOWER($1) LIMIT $2  [["name", "Leadership"], ["LIMIT", 1]]
  SQL (1.6ms)  INSERT INTO "categories" ("name", "created_at", "updated_at") VALU\
ES ($1, $2, $3) RETURNING "id"  [["name", "Leadership"], ["created_at", "2017-07-\
06 04:17:37.419706"], ["updated_at", "2017-07-06 04:17:37.419706"]]
   (0.3ms)  COMMIT
 => #<Category id: 4, name: "Leadership", created_at: "2017-07-06 04:17:37", upda\
ted_at: "2017-07-06 04:17:37">
2.3.3 :012 > Category.find_or_create_by(name: 'Leadership')
  Category Load (0.6ms)  SELECT  "categories".* FROM "categories" WHERE "categori\
es"."name" = $1 LIMIT $2  [["name", "Leadership"], ["LIMIT", 1]]
 => #<Category id: 4, name: "Leadership", created_at: "2017-07-06 04:17:37", upda\
ted_at: "2017-07-06 04:17:37">
```

(the above code snippet online)

Command `:011,Category.find_or_create_by(name: 'Leadership')` tries to locate the `Category` with name equal to `Leadership`. If this `Category` does not exist, it creates it. That is why you see the `INSERT INTO`.

Command `:012` on the other hand, generates only a `SELECT` statement. Rails finds that there is already a `Category` with this name and does nothing by locating and returning it.

Sometimes, the attributes needed to create a resource might be more than the attributes that are used to locate it. In that case, the `find_or_create_by` will fail and will return an invalid instance:

```
2.3.3 :022 >   article = Article.find_or_create_by(title: 'Voice Pitch')
  Article Load (0.5ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."ti\
tle" = $1 LIMIT $2  [["title", "Voice Pitch"], ["LIMIT", 1]]
   (0.2ms)  BEGIN
   (0.2ms)  ROLLBACK
 => #<Article id: nil, title: "Voice Pitch", text: nil, created_at: nil, updated_\
at: nil, user_id: nil, state: "draft">
2.3.3 :023 > article.valid?
 => false
```

(the above code snippet online)

We may get away with this by providing all the attributes in the `find_or_create_by` argument `Hash`:

```
2.3.3 :032 >   article = Article.find_or_create_by(title: 'Voice Pitch', text: 'V\
oice pitch for males and female voices', user: User.first)
  User Load (2.7ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
IT $1  [["LIMIT", 1]]
  Article Load (0.3ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."ti\
tle" = $1 AND "articles"."text" = $2 AND "articles"."user_id" = 10 LIMIT $3  [["t\
itle", "Voice Pitch"], ["text", "Voice pitch for males and female voices"], ["LIM\
IT", 1]]
   (0.2ms)  BEGIN
  SQL (4.2ms)  INSERT INTO "articles" ("title", "text", "created_at", "updated_at\
", "user_id") VALUES ($1, $2, $3, $4, $5) RETURNING "id"  [["title", "Voice Pitch\
"], ["text", "Voice pitch for males and female voices"], ["created_at", "2017-07-\
06 04:24:04.413062"], ["updated_at", "2017-07-06 04:24:04.413062"], ["user_id", 1\
0]]
   (0.3ms)  COMMIT
 => #<Article id: 51, title: "Voice Pitch", text: "Voice pitch for males and fema\
le voices", created_at: "2017-07-06 04:24:04", updated_at: "2017-07-06 04:24:04",\
 user_id: 10, state: "draft">
```

(the above code snippet online)

Here, on command `:032` we provide both `text` and the `user` attributes. So, the `find_or_create_by` succeeds and creates the Article. Potentially, though, this might not be the perfect approach. This is because the above technique will use all the attributes to select an Article. This `WHERE` clause generated might not be optimum, especially when there are no indexes.

The technique that follows is more efficient:

```
2.3.3 :042 > article = Article.create_with(text: 'Most beautiful beaches in Greec\
e are in Cyclades islands ...', user: User.first).find_or_create_by(title: 'Beach\
es in Greece')
  User Load (1.3ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
IT $1  [["LIMIT", 1]]
  Article Load (0.3ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."ti\
tle" = $1 LIMIT $2  [["title", "Beaches in Greece"], ["LIMIT", 1]]
   (0.1ms)  BEGIN
  SQL (1.4ms)  INSERT INTO "articles" ("title", "text", "created_at", "updated_at\
", "user_id") VALUES ($1, $2, $3, $4, $5) RETURNING "id"  [["title", "Beaches in \
Greece"], ["text", "Most beautiful beaches in Greece are in Cyclades islands ..."\
], ["created_at", "2017-07-06 04:27:56.673395"], ["updated_at", "2017-07-06 04:27\
:56.673395"], ["user_id", 10]]
   (0.4ms)  COMMIT
 => #<Article id: 52, title: "Beaches in Greece", text: "Most beautiful beaches i\
n Greece are in Cyclades i...", created_at: "2017-07-06 04:27:56", updated_at: "2\
017-07-06 04:27:56", user_id: 10, state: "draft">
```

(the above code snippet online)

The `.create_with` method call that we call before the `find_or_create_by` will use the `Hash` we give in order to create the `Article`. Hence, the `SELECT` that is generated to check whether the `Article` already exists or not does not take into account the attributes of the `Hash` of the `.create_with`. Only the ones of the `Hash` of the `find_or_create_by`.

Moreover, the `find_or_create_by!` is similar to `find_or_create_by`, but it will raise an exception if the resource is invalid and cannot be created.

```
2.3.3 :046 >   Article.find_or_create_by!(title: 'Trees')
  Article Load (0.5ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."ti\
tle" = $1 LIMIT $2  [["title", "Trees"], ["LIMIT", 1]]
   (0.2ms)  BEGIN
   (0.2ms)  ROLLBACK
ActiveRecord::RecordInvalid: Validation failed: User must exist, User can't be bl\
ank, Text can't be blank
        from (irb):46
```

(the above code snippet online)

Note that sometimes you may want to initialize the non-found resource, without actually saving it in the database. This can be achieved with the `find_or_initialize_by`, which otherwise works similarly to `find_or_create_by`.

Finding by SQL

Sometimes we want to run a custom SQL command. This can by easily achieved with the `find_by_sql`. Here is an example:

```
2.3.3 :050 > articles = Article.find_by_sql("select articles.id as article_id, ar\
ticles.title, users.* from articles join users on users.id = articles.user_id ord\
er by articles.id")
  Article Load (1.1ms)  select articles.id as article_id, articles.title, users.*\
 from articles join users on users.id = articles.user_id order by articles.id
 => [#<Article id: 10, title: "Ruby Arrays", created_at: "2017-06-10 06:46:05", u\
pdated_at: "2017-07-02 09:52:46">, #<Article id: 21, title: "Business Psychology"\
, created_at: "2017-06-11 09:45:22", updated_at: "2017-07-02 09:52:46">, #<Articl\
e id: 21, title: "Unconscious Bias", created_at: "2017-06-11 09:45:22", updated_a\
t: "2017-07-02 09:52:46">, #<Article id: 21, title: "Leadership Authenticity", cr\
eated_at: "2017-06-11 09:45:22", updated_at: "2017-07-02 09:52:46">, #<Article id\
: 18, title: "Team Building", created_at: "2017-06-11 06:58:18", updated_at: "201\
7-07-02 09:52:46">, #<Article id: 18, title: "System 1 and System 2", created_at:\
 "2017-06-11 06:58:18", updated_at: "2017-07-02 09:52:46">, #<Article id: 10, tit\
le: "Beaches in Greece", created_at: "2017-06-10 06:46:05", updated_at: "2017-07-\
02 09:52:46">]
2.3.3 :051 > articles.first.article_id
 => 45
2.3.3 :052 > articles.first.email
 => "mary@gmail.com"
```

(the above code snippet online)

The command `:050` calls `.find_by_sql` giving a custom SQL command. The result is an array of `Article` instances. In fact, it is not exactly like an `Article`. It has the attributes that are specified in the `SELECT` part of the SQL query. That is why you can see that command `:051` retrieves the Article id using the `article_id` attribute and command `:052` retrieves the `email` of the user from what is supposed to be an `Article` instance.

`.pluck`

The `.pluck` method can be used to return an array of the attributes of the records matched. It is similar to `select` but we prefer it when we want the Ruby array of scalar values, rather than instances of the class of the resource

```
2.3.3 :053 > Article.pluck(:id, :title)
   (0.4ms)  SELECT "articles"."id", "articles"."title" FROM "articles"
 => [[46, "Business Psychology"], [47, "Unconscious Bias"], [48, "Leadership Auth\
enticity"], [49, "Team Building"], [50, "System 1 and System 2"], [45, "Ruby Arra\
ys"], [52, "Beaches in Greece"]]
```

(the above code snippet online)

The above could have been implemented with:

```
2.3.3 :055 > Article.select("id, title").map { |article| [article.id, article.tit\
le] }
  Article Load (1.5ms)  SELECT id, title FROM "articles"
 => [[46, "Business Psychology"], [47, "Unconscious Bias"], [48, "Leadership Auth\
enticity"], [49, "Team Building"],
     [50, "System 1 and System 2"], [45, "Ruby Arrays"], [52, "Beaches in Greece"\
]]
```

(the above code snippet online)

So, the `Article.select("id, title").map { |article| [article.id, article.title] }` is equivalent to this: `Article.pluck(:id, :title)`

`.ids`

Often, we only want the ids of the resources, i.e. the primary keys.

```
2.3.3 :056 > Article.ids
   (0.5ms)  SELECT "articles"."id" FROM "articles"
 => [46, 47, 48, 49, 50, 45, 52]
2.3.3 :057 > Article.joins(:user).where(user: User.first).ids
  User Load (0.4ms)  SELECT  "users".* FROM "users" ORDER BY "users"."id" ASC LIM\
IT $1  [["LIMIT", 1]]
   (0.6ms)  SELECT "articles"."id" FROM "articles" INNER JOIN "users" ON "users".\
"id" = "articles"."user_id" WHERE "articles"."user_id" = 10
 => [45, 52]
```

(the above code snippet online)

The command `:056`, `Article.ids` returns an Array with the ids of all the Articles. On the other hand, the command `:057` returns the ids of the Articles of the first user.

Existence of Objects

Checking the existence of rows can be done with the following methods:

1. `exists?`
2. `any?`
3. `many?`

For example:

```
2.3.3 :001 > Article.exists?(45)
  Article Exists (0.7ms)  SELECT  1 AS one FROM "articles" WHERE "articles"."id" \
= $1 LIMIT $2  [["id", 45], ["LIMIT", 1]]
 => true
```

(the above code snippet online)

The `Article.exists?(45)` returns `true` if there is an `Article` with primary key `45`. The argument to `exists?` can be any expression that can be used for `.where` argument too.

Or you can use the `.where` and chain the `.exists?` at the end:

```
2.3.3 :007 > Article.where(title: 'Ruby Arrays').exists?
  Article Exists (0.5ms)  SELECT  1 AS one FROM "articles" WHERE "articles"."titl\
e" = $1 LIMIT $2  [["title", "Ruby Arrays"], ["LIMIT", 1]]
 => true
```

(the above code snippet online)

You can also use `.any?`. For example:

```
2.3.3 :008 > Article.where(title: 'Ruby Arrays').any?
  Article Exists (0.5ms)  SELECT  1 AS one FROM "articles" WHERE "articles"."titl\
e" = $1 LIMIT $2  [["title", "Ruby Arrays"], ["LIMIT", 1]]
 => true
2.3.3 :009 > Article.where(title: 'Ruby Arrays').many?
   (0.5ms)  SELECT COUNT(*) FROM "articles" WHERE "articles"."title" = $1  [["tit\
le", "Ruby Arrays"]]
 => false
2.3.3 :014 > Article.where(title: 'Ruby Arrays').count
   (0.5ms)  SELECT COUNT(*) FROM "articles" WHERE "articles"."title" = $1  [["tit\
le", "Ruby Arrays"]]
 => 1
```

(the above code snippet online)

You can see above that with 1 Article matching the where clause, the `.any?` returns `true`, but the `many` returns `false`.

So, `.many?` returns `true` if there are at least 2.

Calculations

We have already seen the `.count` call that is translated to `SELECT COUNT(*)` in the database.

Other methods that are translated to corresponding calculation functions in SQL are:

1. `.average`
2. `.minimum`
3. `.maximum`
4. `.sum`

Models that are not Active Records

Models do not have to be Active Records. We have already used a model that is not Active Record. It is the `SignIn` model. You can see that it is a simple class that includes the `ActiveModel::Model` module. This allows you to define validations like you did for the Active Record models.

`ActiveModel::Model` is very convenient and powerful. It is used when we want to model a domain object that does not necessarily have a persistent state saved in the database.

Closing Note

That was a long chapter with many concepts visited and revisited. You should now feel more confident with the Rails Models, but, admittedly, you will conquer the ideas only if you practice.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz Details

The quiz for this chapter can be found here

Task Details

Make sure that the following features/tasks, that have been explained in the chapter, have been accomplished in your `blog` application.

1. The Sign Up Form should display the terms of service. The user should confirm that they accept the terms of service
2. Make sure that whenever an User is deleted, their corresponding Articles are deleted too.
3. Introduce the model `Category` and make sure that there is a many-to-many association between Articles and Categories.
4. Make also use of the `has_many through` association so that you can travel directly from an article to its categories.
5. Make sure that the index page for Articles uses `find_each` to parse all the Articles.
6. Make sure that the Articles are sorted by `title` when listed in the index page.

8 - RoR - Ruby Language Extensions

Summary

RoR Ruby Language Extensions - Summary

Ruby on Rails has enriched Ruby language itself with methods and utilities that will ease your work of Web development. These extensions are also used internally by RoR framework itself. You need to study the API because it will be proven absolutely useful to you.

Learning Goals

1. Learn about the `blank?` and `present?` methods.
2. Learn about the `presence` method.
3. Learn about the `try` method.
4. Learn about the `to_param` method.
5. Learn about the `to_query` method.
6. Learn about the `in?` method.
7. Learn about the way to delegate a method call to another object.
8. Learn about how to define class attributes that can be overridden in subclasses.
9. Learn how you can get a list of subclasses of a class.
10. Learn how you can get a list of descendants of a class.
11. Learn about extensions to `String` class, like the method `#remove`.
12. Learn about extensions to `Numeric` class, like the method `#megabytes`.
13. Learn about methods like `#minutes` and `#hours`.
14. Learn about extensions to `Enumerable` module, like the method `#sum`.
15. Learn about extensions to the `Hash` class.
16. Learn about extensions to `Date` and `DateTime` classes.
17. Learn where you can find more information about the `ActiveSupport` extensions.

Introduction

Ruby on Rails has extended Ruby language with some methods that make our lives easier and support the concepts and ideas that RoR tries to convey. This enrichment is meant to be used both by Ruby on Rails applications and by the framework itself.

Let's see some of the most important ones:

Extensions To All Objects

`blank?` and `present?`

The `#blank?` method returns `true` in the following cases:

1. When it is called on something that is `nil`. In other words `nil.blank?` is `true`:

```
2.3.3 :001 > nil.blank?
 => true
```

2. When it is called on something that is `false`:

```
2.3.3 :002 > false.blank?
 => true
```

3. When it is called on strings that are composed only of whitespace characters or that they are completely empty:

```
2.3.3 :003 > ''.blank?
 => true
2.3.3 :004 > ' '.blank?
 => true
2.3.3 :005 > "     \n    ".blank?
 => true
```

4. When it is called on empty arrays:

```
2.3.3 :006 > [].blank?
 => true
```

5. When it is called on empty hashes:

```
2.3.3 :007 > {}.blank?
 => true
```

6. When it is called on an object that responds to `#empty?` and returns `true`:

```
2.3.3 :004 > class A
2.3.3 :005?>   def empty?
2.3.3 :006?>     true
2.3.3 :007?>     end
2.3.3 :008?>   end
 => :empty?
2.3.3 :009 > a = A.new
 => #<A:0x007fe6f2148b48>
2.3.3 :010 > a.blank?
 => true
2.3.3 :011 > class B
2.3.3 :012?>   def empty?
```

```
2.3.3 :013?>     false
2.3.3 :014?>     end
2.3.3 :015?>   end
 => :empty?
2.3.3 :016 > b = B.new
 => #<B:0x007fe6eb6660c8>
2.3.3 :017 > b.blank?
 => false
```

As you can see above, instance of `A` class is blank whereas instance of `B` class is not blank. This is because the instances of `A` return `true` on `#empty?` call. Whereas instances of `B` return `false` on `#empty?` call.

Note that the `present?` is implemented as `!blank?`. For example `false.present?` returns `false`, which catches by surprise the novice Ruby on Rails developers.

presence

The `#presence` returns its receiver if the receiver `#present?` returns `true`. Otherwise, it returns `nil`. So, instead of writing

```
foo = bar.present? ? bar : 'foo'
```

(the above code snippet online)

you can write

```
foo = bar.presence || 'foo'
```

(the above code snippet online)

try

There are many times in which you want to call a method on an object, but you don't know whether the object is `nil` or not. Hence, you end up writing code like this:

```
if @invoice.present?
  @invoice.print
end
```

(the above code snippet online)

With `try`, you can call the method and it will not fail if the receiver is `nil`:

```
@invoice.try(:print)
```

(the above code snippet online)

You can also call `try` and pass a block. This is useful if you want to call many methods on the receiver that could potentially be `nil`:

```ruby
@invoice.try do |invoice|
  invoice.print
  invoice.store
end
```

(the above code snippet online)

However, in the latter case, you should prefer the `try!` instead, because it will catch your spelling errors, whereas the `try` would swallow them.

to_param

Sometimes you want to include an object into query string parameters. In order to do that Rails calls the method `#to_param` to that object. The result is being used as the *value* part of the query string for this parameter or as URL fragment.

```
2.3.3 :001 > "Foo".to_param
 => "Foo"
2.3.3 :002 > "this is hello world".to_param
 => "this is hello world"
```

(the above code snippet online)

As you can see the value is not URL encoded.

The idea here is that you may want to override the `#to_param` method for particular classes. For example:

```
2.3.3 :003 > class A
2.3.3 :004?>   def initialize(day_of_week)
2.3.3 :005?>     @day_of_week = day_of_week
2.3.3 :006?>     end
2.3.3 :007?>   def to_param
2.3.3 :008?>     case @day_of_week
2.3.3 :009?>       when 0
2.3.3 :010?>         return 'Sunday'
2.3.3 :011?>       when 1
2.3.3 :012?>         return 'Monday'
2.3.3 :013?>       when 2
2.3.3 :014?>         return 'Tuesday'
2.3.3 :015?>       when 3
2.3.3 :016?>         return 'Wednesday'
2.3.3 :017?>       when 4
2.3.3 :018?>         return 'Thursday'
2.3.3 :019?>       when 5
2.3.3 :020?>        return 'Friday'
2.3.3 :021?>       when 6
```

```
2.3.3 :022?>         return 'Saturday'
2.3.3 :023?>       else
2.3.3 :024 >         raise 'Not valid day of week'
2.3.3 :025?>       end
2.3.3 :026?>     end
2.3.3 :027?>   end
 => :to_param
2.3.3 :028 > a = A.new(3)
 => #<A:0x007fe6eaed9080 @day_of_week=3>
2.3.3 :029 > a.to_param
 => "Wednesday"
```

(the above code snippet online)

Then, when, for example, you call route url helpers, that take arguments of class `A`, the values will be converted using `#to_param` in order to build the query string and/or the URL.

to_query

The `#to_query` can be used to build both the key and the value to be used in a query string. Also, both the key and the value will be URL encoded. If we continue the previous example...

```
2.3.3 :030 > a.to_query('foo')
 => "foo=Wednesday"
2.3.3 :031 > a.to_query('fantastic day')
 => "fantastic+day=Wednesday"
```

(the above code snippet online)

in?

The method `#in?` checks whether the receiver is included in the list of arguments given to the method. So, instead of writing:

```
[1, 2, 3].include?(1)
```

(the above code snippet online)

we write

```
in?([1, 2, 3])
```

(the above code snippet online)

Extensions To `Module`

Method Delegation

You can use the method `delegate` in order to make sure that the implementation of a method is offered by another object and not by the receiver.

For example:

```
class Invoice < ActiveRecord
  belongs_to :user

  delegate :email, address, to: :user
  ...
end
```

(the above code snippet online)

In that case, I can have an `Invoice` instance and call `#email` on it. The call will be forwarded to the `user` associated to the invoice at hand.

Extensions to `Class`

`class_attribute`

The `class_attribute` allows you to define attributes at `Class` level. These attributes can be overridden by sub-classes.

The *issue* with class attributes that come with Ruby is that changing the value of a class attribute via a sub-class affects the value of the class attribute at the superclass too:

```
2.3.3 :001 > class A
2.3.3 :002?>   cattr_accessor :a
2.3.3 :003?>   end
 => [:a]
2.3.3 :004 > class B < A
2.3.3 :005?>   end
 => nil
2.3.3 :006 > A.a = 5
 => 5
2.3.3 :007 > B.a
 => 5
2.3.3 :008 > B.a = 4
 => 4
2.3.3 :009 > A.a
 => 4
```

(the above code snippet online)

As you can see, the command `:008` affects the value of the `a` class attribute for the `A` class too. If you want to have inheritance with attributes that are independent, you should use the `class_attribute` instead:

```
2.3.3 :001 > class A
2.3.3 :002?>   class_attribute :a
2.3.3 :003?>   end
 => [:a]
2.3.3 :004 > class B < A
2.3.3 :005?>   end
 => nil
2.3.3 :006 > A.a = 5
 => 5
2.3.3 :007 > B.a
 => 5
2.3.3 :008 > B.a = 4
 => 4
2.3.3 :009 > A.a
 => 5
2.3.3 :010 > B.a
 => 4
```

(the above code snippet online)

You can see that the command `:008` does not affect the `A.a` value.

subclasses

The `subclasses` method returns the list of classes that are subclasses (direct descendants) of the receiver:

```
2.3.3 :003 > A.subclasses
 => []
2.3.3 :004 > class B < A
2.3.3 :005?>   end
 => nil
2.3.3 :006 > A.subclasses
 => [B]
2.3.3 :007 > class C < B
2.3.3 :008?>   end
 => nil
2.3.3 :009 > A.subclasses
 => [B]
```

(the above code snippet online)

descendants

The `descendants` method returns the list of classes that are subclasses of the receiver but not necessarily direct subclasses:

```
2.3.3 :001 > class A
2.3.3 :002?>   end
 => nil
2.3.3 :003 > A.subclasses
 => []
2.3.3 :004 > class B < A
2.3.3 :005?>   end
 => nil
2.3.3 :006 > A.subclasses
 => [B]
2.3.3 :007 > class C < B
2.3.3 :008?>   end
 => nil
2.3.3 :009 > A.subclasses
 => [B]
2.3.3 :010 > A.descendants
 => [C, B]
```

(the above code snippet online)

You can see that `C` is part of the `descendants` but not part of the `subclasses` of `A`.

Extensions to String

remove

The method `remove` is added to `String` instances. It removes all the occurrences of the pattern given as argument:

```
2.3.3 :011 > "Hello World!".remove /d!/
 => "Hello Worl"
```

(the above code snippet online)

squish

It removes leading and trailing whitespace, as well as it replaces multiple sequences of whitespace with single space.

```
2.3.3 :012 > "     \n \n    Hello        World! I \n\n     love Ruby!    \n\n    \
\n    ".squish
 => "Hello World! I love Ruby!"
```

(the above code snippet online)

truncate

Returns the first part of the receiver up to the number of characters given as input argument. Note that the last 3 characters are replaced with . . . :

```
2.3.3 :014 > "This is a long sentence!".truncate(10)
 => "This is..."
```

(the above code snippet online)

This method is very useful in views when we have limited width to display a long value.

from

Returns the substring of the string starting at position given as input argument:

```
2.3.3 :015 > "This is a long sentence!".from(5)
 => "is a long sentence!"
```

(the above code snippet online)

to

Returns the first part of the string up to the position specified by the input argument:

```
2.3.3 :017 > "This is a long sentence!".to(4)
 => "This "
```

(the above code snippet online)

camelize

It turns a string to its Camel format equivalent. It is usually used to strings that have the snake format.

```
2.3.3 :019 > "invoice_item".camelize
 => "InvoiceItem"
```

(the above code snippet online)

underscore

It does the opposite of `camelize`:

```
2.3.3 :020 > "InvoiceItem".underscore
 => "invoice_item"
```

(the above code snippet online)

titleize

It capitalizes the words in the receiver.

```
2.3.3 :021 > "hello world".titleize
 => "Hello World"
```

(the above code snippet online)

parameterize

The string is normalized so that it can be used in URLs:

```
2.3.3 :023 > "Hello World".parameterize
 => "hello-world"
```

(the above code snippet online)

tableize

It turns a string to a string that can be used as a table name. It's useful when converting a class name to a table name:

```
2.3.3 :025 > "InvoiceItem".tableize
 => "invoice_items"
```

(the above code snippet online)

classify

It turns a table name to a name that can be used as a `Class` name:

```
2.3.3 :026 > "invoice_items".classify
 => "InvoiceItem"
```

(the above code snippet online)

constantize

It returns its receiver to a constant and resolves it, which means it needs to be part of the defined constants.

```
Loading development environment (Rails 5.1.1)
2.3.3 :001 > class MyClass
2.3.3 :002?>   end
 => nil
2.3.3 :003 > "MyClass".constantize.new
 => #<MyClass:0x007fe6f22f8c40>
2.3.3 :004 > "NonExistingClass".constantize
NameError: uninitialized constant NonExistingClass
        from (irb):4
```

(the above code snippet online)

This method can be really useful when coding at meta-programming level.

humanize

It converts a string to something that can be easier read by a human:

```
2.3.3 :008 > 'comments_count'.humanize
 => "Comments count"
```

(the above code snippet online)

foreign_key

It takes a class name and returns the foreign key column name:

```
2.3.3 :009 > 'Customer'.foreign_key
 => "customer_id"
```

(the above code snippet online)

to_date, to_time, to_datetime

These methods convert the string receiver to a `Date`, or `Time` or `DateTime` instance.

```
2.3.3 :010 > "2017-07-23 12:04:04".to_datetime
 => Sun, 23 Jul 2017 12:04:04 +0000
2.3.3 :012 > "2017-07-23 12:04:04".to_time(:local)
 => 2017-07-23 12:04:04 +0300
2.3.3 :013 > "2017-07-23 12:04:04".to_time(:utc)
 => 2017-07-23 12:04:04 UTC
```

(the above code snippet online)

See how the `to_time` takes as argument either `:local` or `:utc` and converts the string into the local or UTC time respectively.

Extensions to Numeric

Bytes

All numbers now respond to the following methods:

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

(the above code snippet online)

For example:

```
2.3.3 :016 > 3.megabytes
 => 3145728
2.3.3 :017 > 3 * 1024.kilobytes
 => 3145728
```

(the above code snippet online)

Time

You can do time calculations since numbers respond to methods like `#minutes`, `#hours`, `#years`, `#seconds` and more:

```
2.3.3 :018 > 24.minutes.from_now
 => Sun, 23 Jul 2017 09:55:07 UTC +00:00
2.3.3 :019 > 30.minutes + 30.minutes == 1.hour
 => true
2.3.3 :021 > Time.zone.now
 => Sun, 23 Jul 2017 09:32:02 UTC +00:00
2.3.3 :022 > Time.zone.now - 2.hours
 => Sun, 23 Jul 2017 07:32:04 UTC +00:00
```

(the above code snippet online)

Formatting

You can use various methods to format numbers.

For example:

```
2.3.3 :023 > 1024.to_s(:human_size)
 => "1 KB"
```

(the above code snippet online)

Extensions to Enumerable

sum

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

```
2.3.3 :024 > [1, 2, 3].sum
 => 6
```

(the above code snippet online)

many?

The method `many?` returns `true` if there are more than one element in the collection:

```
2.3.3 :025 > [1, 2, 3].many?
 => true
2.3.3 :026 > [1].many?
 => false
2.3.3 :027 > [].many?
 => false
```

(the above code snippet online)

exclude?

It returns `true` if an element does not belong to a collection:

```
2.3.3 :028 > [1, 2, 3].exclude?(2)
 => false
2.3.3 :029 > [1, 2, 3].exclude?(4)
 => true
```

(the above code snippet online)

without

It returns a collection copy without the specified elements in:

```
2.3.3 :030 > [1, 2, 3, 4].without(2, 3)
 => [1, 4]
```

(the above code snippet online)

Extensions to Hash

Merging

Ruby already supports Hash merging:

```
2.3.3 :033 > {a: '1', b: '2', c: '0'}.merge({c: '3', d: '4'})
 => {:a=>"1", :b=>"2", :c=>"3", :d=>"4"}
```

(the above code snippet online)

but Active Support adds some useful extensions.

For example, the `reverse_merge`. With `reverse_merge` when there is a key collision the winner is the receiver and not the Hash which is given as argument.

`except`

It returns a Hash with the keys in the arguments removed:

```
2.3.3 :035 > {a: '1', b: '2', c: '0'}.except(:a, :c)
 => {:b=>"2"}
```

(the above code snippet online)

The `except!` updates the receiver:

```
2.3.3 :036 > h = {a: '1', b: '2', c: '3'}
 => {:a=>"1", :b=>"2", :c=>"3"}
2.3.3 :037 > h.except!(:b)
 => {:a=>"1", :c=>"3"}
2.3.3 :038 > h
 => {:a=>"1", :c=>"3"}
2.3.3 :039 >
```

(the above code snippet online)

`transform_keys`

The `transform_keys` converts keys using a logic given as a block to the method call:

```
2.3.3 :041 > {a: '1', b: '2', c: '3'}.transform_keys { |k| k.to_s.upcase }
 => {"A"=>"1", "B"=>"2", "C"=>"3"}
```

(the above code snippet online)

The `transform_keys!` updates the receiver.

Also, the `deep_transform_keys` does a deep operation on nested hashes:

```
2.3.3 :046 > {a: '1', b: '2', c: '3', d: {foo: :bar}}.transform_keys { |k| k.to_s\
.upcase }
 => {"A"=>"1", "B"=>"2", "C"=>"3", "D"={:foo=>:bar}}
2.3.3 :047 > {a: '1', b: '2', c: '3', d: {foo: :bar}}.deep_transform_keys { |k| k\
.to_s.upcase }
 => {"A"=>"1", "B"=>"2", "C"=>"3", "D"={"FOO"=>:bar}}
2.3.3 :048 >
```

(the above code snippet online)

`stringify_keys`

It converts the keys of a Hash to their string representation by calling `#to_s` on each one of them:

```
2.3.3 :042 > {a: '1', b: '2', c: '3'}.stringify_keys
 => {"a"=>"1", "b"=>"2", "c"=>"3"}
```

(the above code snippet online)

The `stringify_keys!` updates the receiver.

Also, the `deep_stringify_keys` does a deep conversion of the keys to strings:

```
2.3.3 :048 > {a: '1', b: '2', c: '3', d: {foo: :bar}}.stringify_keys
 => {"a"=>"1", "b"=>"2", "c"=>"3", "d"={:foo=>:bar}}
2.3.3 :049 > {a: '1', b: '2', c: '3', d: {foo: :bar}}.deep_stringify_keys
 => {"a"=>"1", "b"=>"2", "c"=>"3", "d"={"foo"=>:bar}}
```

(the above code snippet online)

`symbolize_keys`

It converts the keys to their symbol equivalent by calling `#to_sym` on each one of them:

```
2.3.3 :051 > {'a' => '1', 'b' => '2', 'c' => '3', 'd' => {'foo' => :bar}}.symboli\
ze_keys
 => {:a=>"1", :b=>"2", :c=>"3", :d={"foo"=>:bar}}
```

(the above code snippet online)

The `symbolize_keys!` updates the receiver.

Also, the `deep_symbolize_keys` does a deep symbolization of the keys:

```
2.3.3 :052 > {'a' => '1', 'b' => '2', 'c' => '3', 'd' => {'foo' => :bar}}.deep_sy\
mbolize_keys
 => {:a=>"1", :b=>"2", :c=>"3", :d=>{:foo=>:bar}}
```

(the above code snippet online)

`assert_valid_keys`

This is very useful to make sure that a Hash holds only valid keys:

```
2.3.3 :057 >   {username: 'foo', password: 'bar'}.assert_valid_keys(:username, :p\
assword)
 => {:username=>"foo", :password=>"bar"}
2.3.3 :058 > {username: 'foo', password: 'bar', type: 'customer'}.assert_valid_ke\
ys(:username, :password)
ArgumentError: Unknown key: :type. Valid keys are: :username, :password
        from (irb):58
```

(the above code snippet online)

As you can see, the second call to `assert_valid_keys` raises an error of type `ArgumentError` telling us that the Hash contains keys that are not valid (`:type` key is not in the white list of keys).

Indifferent Access

The method `with_indifferent_access` returns a hash which is of type `ActiveSupport::HashWithIndifferentAc` This has the ability to treat symbol and string keys as equal:

```
2.3.3 :063 > {a: 1, b: 2}.has_key?('a')
 => false
2.3.3 :064 > {a: 1, b: 2}.with_indifferent_access.has_key?('a')
 => true
```

(the above code snippet online)

Extensions to Date

`Date.current`

It returns the date of today in current time zone.

`Date.today`

It returns the date of today but does not honour the current time zone.

`Date.yesterday`, `Date.tomorrow`

They return the date before today and the date after today respectively.

`past?`, `today?`, `future?`, `on_weekday?`, on_weekend¿

These are methods that should be called on a `Date` instance:

```
2.3.3 :018 > Date.current
 => Sun, 23 Jul 2017
2.3.3 :019 > Date.current.past?
 => false
2.3.3 :020 > (Date.current - 1.day).past?
 => true
2.3.3 :021 > (Date.current - 1.day).today?
 => false
2.3.3 :022 > (Date.current - 1.day).future?
 => false
2.3.3 :023 > (Date.current + 1.day).future?
 => true
2.3.3 :024 > (Date.current + 1.day).past?
 => false
2.3.3 :025 > Date.current.on_weekend?
 => true
2.3.3 :026 > (Date.current + 3.days).on_weekday?
 => true
2.3.3 :027 > (Date.current + 3.days).on_weekend?
 => false
```

(the above code snippet online)

Named Dates

ActiveSupport provides methods like `#prev_year` and `#next_year` that calculate another Date based on a Date at hand.

Extensions to DateTime

ActiveSupport defines extensions to DateTime. For example `DateTime.current` returns the current date and time using the currently defined time zone.

Closing Note

ActiveSupport adds a lot of useful methods which you can use to your own benefit in your Ruby on Rails applications. You can read more about it in the Rails Guides here.

Tasks and Quizzes

Before you continue, you may want to know that: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz Details

The quiz for this chapter can be found here

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

9 - Internationalizing Your Application

Summary

Internationalization With Rails - Summary

Although English is a very popular language, we would like to make our Web application to support multiple languages, like French, Spanish, German, Chinese and more. Rails has fantastic support for internationalizing your application, both with regards to the strings that need to be translated, but also with regards to how dates and currencies should be displayed. Also, Rails allows you to display time related information using your Users' time zone, which is an awesome feature. All these things you are going to learn and practice in this chapter. You will apply them to your blog application.

Learning Goals

1. Learn about what *internationalization* is.
2. Learn which are the most important methods of the gem that gives the capability for internationalization.
3. Learn how you can get the current locale.
4. Learn how you can get the list of available locales.
5. Learn what you have to do in order to support multiple locales.
6. Learn how you can get the default locale.
7. Learn how to change the default locale of your application.

8. Learn how you can set the locale per request.
9. Learn how to ask Rails to translate a string on a view.
10. Learn how you can set the locale dynamically.
11. Learn how to set the locale from the params.
12. Learn how to set the default url options for all your controllers.
13. Learn how to set the locale based on the user browser settings on preferred language.
14. Learn how to set the locale based on the user preferences stored in your database.
15. Learn how to build the UI to allow user to set their preferred language.
16. Learn how you can build different select boxes to support preferred language and time zone.
17. Learn how to submit a form when the value of its input changes.
18. Learn how to redirect back to where the user came from and tune the params of the redirect URL.
19. Learn how to create translation strings that have variable parts.
20. Learn how to display timestamps in views by taking into account the locale selected.
21. Learn how to define keys nested in namespaces.
22. Learn how to define different time formats.
23. Learn how to localize the validation error messages.
24. Learn how to list all the supported time zones.
25. Learn how to set the application default time zone.
26. Learn how to get the current time in the current time zone.
27. Learn how to convert the current time from current time zone to utc.
28. Learn how to display times in the time zone of the user.

Introduction

Internationalization (i18n for short) is the process of making your application support multiple languages. It is not only how an English text is translated to another language text, but it is also how dates, for example, are being displayed too.

The gem that gives us the capability to make our application support multiple languages is the `i18n` gem.

The Most Important Methods

The most important methods are the `translate` and the `localize` method.

On your `blog` application root folder, start a Rails console and try the following commands:

```
2.3.3 :001 > I18n.translate 'hello'
 => "Hello world"
2.3.3 :002 > I18n.localize Date.current
 => "2017-07-23"
2.3.3 :003 > Date.current
 => Sun, 23 Jul 2017
```

(the above code snippet online)

The `translate` method takes a string and returns its translation in the current locale. The `localize` takes a `Date` or `DateTime` or `Time` object and returns its value in the current locale.

These methods are so important that they have an alias:

```
2.3.3 :007 > I18n.t 'hello'
 => "Hello world"
2.3.3 :008 > I18n.l Date.current
 => "2017-07-23"
```

(the above code snippet online)

Getting Current Locale

The current locale is retrieved with the call `I18n.locale`:

```
2.3.3 :011 > I18n.locale
 => :en
```

(the above code snippet online)

Getting Available Locales

The available locales, i.e. the locales that your application supports are listed with the `I18n.available_locales`:

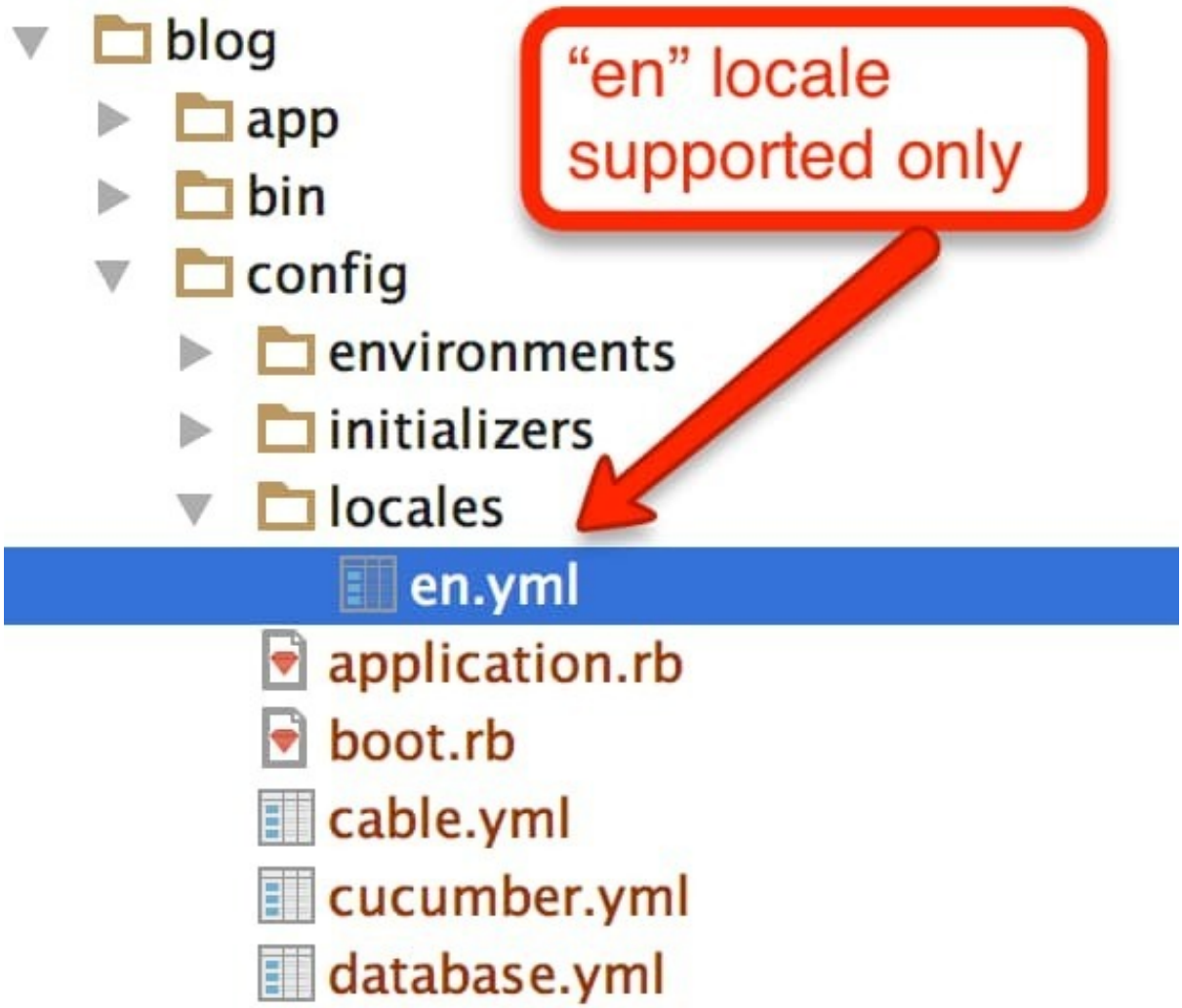
```
2.3.3 :014 > I18n.available_locales
 => [:en]
```

(the above code snippet online)

As you can see, our `blog` application supports only one locale at the moment, the `:en` locale.

Supporting Another Locale

In order to support another locale, you have to create the corresponding YML file inside the folder `config/locales`. Currently, there is only one, the `en.yml` file:



en Locale Is The Only One Supported By Default

That is why the `I18n.available_locales` returns only `:en`.

Let's add support for Spanish. Create the file `config/locales/es.yml` with the following content:

```
# File: config/locales/es.yml
#
es:
  hello: "Hola Mundo"
```

(the above code snippet online)

After you save this file, restart your Rails console and try the following commands:

```
2.3.3 :001 > I18n.available_locales
 => [:en, :es]
2.3.3 :002 > I18n.translate 'hello'
 => "Hello world"
2.3.3 :003 > I18n.locale = :es
 => :es
2.3.3 :004 > I18n.translate 'hello'
 => "Hola Mundo"
```

(the above code snippet online)

On command `:003` we change the current locale to be `:es`. Then, when we call `I18n.translate` we get the corresponding entry from the `config/locales/es.yml` file.

The Default Locale

When a Rails application starts, it has a default locale. You can get it with `I18n.default_locale`:

```
2.3.3 :001 > I18n.default_locale
 => :en
```

(the above code snippet online)

If you want to change the default locale the application starts with, you have to do that inside the `config/application.rb` file:

```
1  # File: config/application.rb
2  #
3  require_relative 'boot'
4  ...
5
6  module Blog
7    class Application < Rails::Application
8      # Initialize configuration defaults for originally generated Rails version.
9  ...
10
11     config.i18n.default_locale = :es
12   end
13 end
```

(the above code snippet online)

Do you see line 21? This is how we configure the `Blog` application to have default locale equal to `:es`.

Now if you execute:

```
blog $ bin/rails runner "puts I18n.locale"
es
blog $
```

(the above code snippet online)

You can see that the local is `es` and this is because the default locale has been set to `:es` inside the `config/application.rb` file.

Let's put the default locale back to `:en`:

```
# File: config/application.rb
#
require_relative 'boot'
...

module Blog
  class Application < Rails::Application
    # Initialize configuration defaults for originally generated Rails version.
...

    config.i18n.default_locale = :en # although setting it to :en is not necessar\
y. This is what Rails considers as default locale anything.
  end
end
```

(the above code snippet online)

Setting The Locale Per Request

Each HTTP Request can be served with its own locale as long as the code sets the `I18n.locale` value at the start of processing the request.

Let's see an example:

Add Translations For `List Of Articles`

Let's suppose that we want the label `List Of Articles` to be displayed according to the locale set for a request. In that case, we have to make the `List Of Articles` being printed on page using the `I18n.t` method. Here it is how:

```
1 - # File: app/views/articles/index.html.haml
2 - #
3 %h1= I18n.t 'list_of_articles'
4 %table.table.table-striped
5 ...
6     = link_to 'Home', root_path
```

(the above code snippet online)

Do you see line 3? It is using `I18n.t` to fetch the translation of the `list_of_articles` key from the database of translations, i.e. from either `config/locales/en.yml` or `config/locales/es.yml`.

Then, make sure that the `config/locales/en.yml` and `config/locales/es.yml` have the key `list_of_articles` listed with the corresponding correct translation:

```
# File: config/locales/en.yml
#
en:
  hello: "Hello World"
  list_of_articles: "List of Articles"
```

(the above code snippet online)

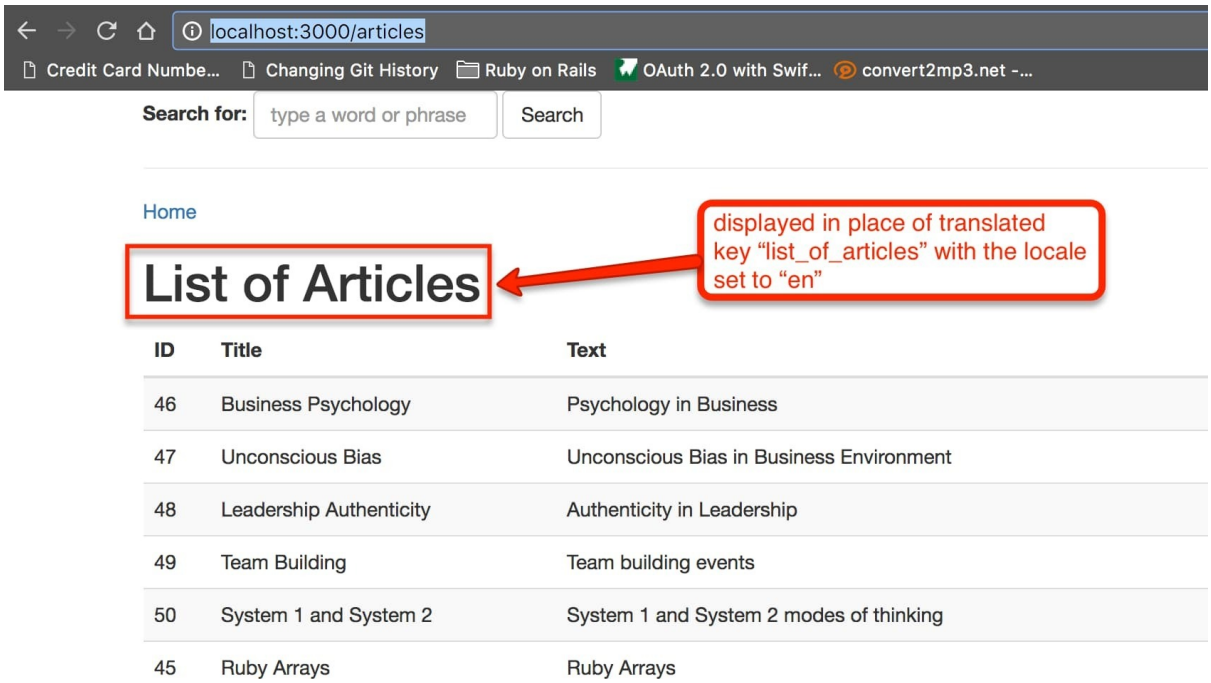
and

```
# File: config/locales/es.yml
#
es:
  hello: "Hola Mundo"
  list_of_articles: "Lista de artículos"
```

(the above code snippet online)

> *Hint:* Restart your server in order to make sure that it reads your latest changes from the file `config/application.rb` which sets the default locale to `en`. Restarting your server is necessary if you change the `config/application.rb` file because it is loaded at the beginning and it is not reloaded on every request.

Now visit the page http://localhost:3000/articles. You will see the page with the `List of Articles` displayed in the place of `= I18n.t 'list_of_articles'`:



List Of Articles Displayed with en Locale

Hint: Note that inside the views, you can use the view helper `t` instead of `I18n.t`:`haml - # File: app/views/articles/index.html.haml - # %h1= t 'list_of_articles' %table.table.table-striped ... = link_to 'Home', root_path` (the above code snippet online)

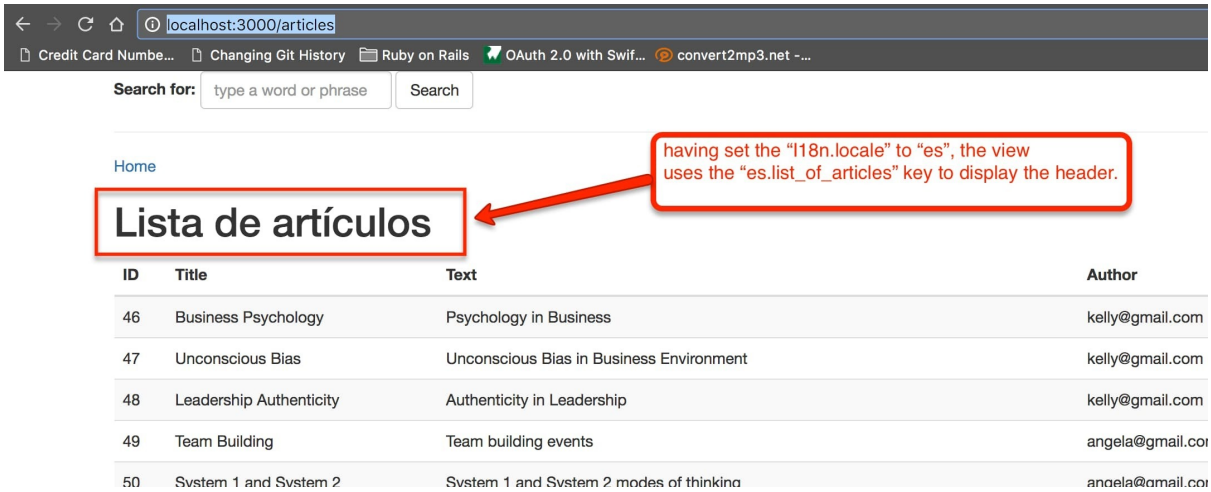
Set the Locale

Now, let's set the locale to be `:es` at the beginning of the request. In order to do that, you can add a `before_action` that would set the `I18n.locale`. Let's do that inside the `ApplicationController` which is the controller that every controller derives from, hence, the `before_action` will affect all the requests.

```
# File: app/controllers/application_controller.rb
#
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  before_action :authenticate_user
  before_action :set_locale

...
  def set_locale
    I18n.locale = :es
  end
end
```

(the above code snippet online)

Having done this change, let's visit the page http://localhost:3000/articles again. You will see this:



es Locale Is Used to Display the List Of Articles header

Setting The Locale Dynamically

Hard coding the locale to `:es`, as you can understand, is not something that works in a production application. There are various ways you can set the locale dynamically. Here are some approaches:

1. Set the locale based on the user language preferences. The `Accept-Language` header indicates which is the preferred browsing language for the browser the user is using.
2. Set the locale based on the Top-Level-Domain value (TLD). For example, the locale can be `es` if the host requested is `www.myawesomeapp.com.es`.
3. Set the locale based on the sub-domain. For example, the locale can be `es` if the host requested is `es.myawesomeapp.com`.
4. Set the locale based on a URL path part. For example, the locale can be `es` if the URL requested is `www.myawesomeapp.com/es/`.
5. Set the locale based on a URL query string param. For example, the locale can be `es` if the URL requested is `www.myawesomeapp.com/articles?locale=es`.
6. Set the locale based on user preferences that you have set in your database against the `User` model.

Sometimes we use a combination of the above. For example, one good strategy is the following:

1. If the user is logged in, then set the locale based on their locale.
2. If the user is not logged in, but the params hash contains the locale, then use whatever the value is specified.
3. Otherwise, use whatever is specified in the `Accept-Language` request header.
4. In any other case, use the default locale specified by the `I18n.default_locale`.

Get the locale from the `params`

Let's adapt our code to be fetching the locale from the `params`. If the `params` does not contain any locale specification, for the time being, we will be selecting the default locale:

```ruby
# File: app/controllers/application_controller.rb
#
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  before_action :authenticate_user
  before_action :set_locale

...
  def set_locale
    locale_requested = params[:locale]
    if locale_requested && locale_requested.to_sym.in?(I18n.available_locales)
      I18n.locale = locale_requested
    else
      I18n.locale = I18n.default_locale
    end
  end
end
```

(the above code snippet online)

Do you see the new implementation of `set_locale` method? It tries to set the locale according to what `params[:locale]` value is. Note also that it checks whether the locale specified, if any, is included in the `available_locales`.

Save the above and try to visit the page http://localhost:3000/articles?locale=en. You will see the `List of Articles` header. Whereas if you visit the http://localhost:3000/articles?locale=es, you will see the `Lista de artículos` header.

Also, if you try to visit the page with a locale that is not supported, you will see the `List of Articles` header, using the default `:en` locale: http://localhost:3000/articles?locale=de.

Get the locale from the `params` When In URL path

The `params` will give you the `locale` even if it is part of the URL path, and not only if it is part of the query string params. However, it does not work just like that. You will need to define your `config/routes.rb` accordingly and, also, configure your `ApplicationController` so that it includes the `locale` in the default URL params, useful when using the route URL helpers, like `articles_path` for example.

Firstly, we have to define an optional scope inside our `config/routes.rb` file:

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  scope "(:locale)", locale: Regexp.new(I18n.available_locales.map(&:to_s).join('\
|')) do
    resources :articles do
      collection do
        get :foo
        get :bar
      end
    end

    get '/sign_up', to: 'sign_up#new'
    post '/sign_up', to: 'sign_up#create'
    get '/confirm_email_address', to: 'sign_up#confirm_email_address' # ?email=<e\
mail>&email_confirmation_token=<email_confirmation_token>
    post '/confirm_email_address_resend', to: 'sign_up#confirm_email_address_rese\
nd'

    get '/sign_in', to: 'sign_in#new'
    post '/sign_in', to: 'sign_in#create'

    get '/sign_out', to: 'sign_in#destroy'
    delete '/sign_out', to: 'sign_in#destroy'
    root 'welcome#index'
  end
end
```

(the above code snippet online)

This is the same `config/routes.rb` file, like before. However, we have enclosed all the route definitions inside the

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  scope "(:locale)", locale: Regexp.new(I18n.available_locales.map(&:to_s).join('\
|')) do
...
  end
end
```

(the above code snippet online)

The `scope...` makes sure that you can optionally have the locale as first part in your URL paths. So,

- `http://localhost:3000/es/articles` is going to set the `params[:locale]` to `es`.

- `http://localhost:3000/es` is going to set the `params[:locale]` to `es`.
- `http://localhost:3000/articles` is going to set the `params[:locale]` to nothing but thanks to the `set_locale` implementation, the locale selected will the the default locale.

and so on.

Secondly, you will have to set the default URL options at the `ApplicationController` level:

```
# File: app/controllers/application_controller.rb
#
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  before_action :authenticate_user
  before_action :set_locale

...

  def default_url_options
    { locale: I18n.locale }
  end

  def set_locale
    locale_requested = params[:locale]
...
end
```

(the above code snippet online)

Do you see lines 27 to 29? It is the implementation of the `default_url_options` method, which now tells that `locale` is going to be in by default.

Having done that, try to visit the following pages:

- http://localhost:3000/articles will display the header `List of Articles`.
- http://localhost:3000/es/articles will display the header `Lista de artículos`.

Nice!

Get the Locale from Accept-Language

Assuming that `params[:locale]` does not specify the locale, let's try to get it from the `Accept-Language` then:

```
# File: app/controllers/application_controller.rb
#
class ApplicationController < ActionController::Base
...
  def default_url_options
    { locale: I18n.locale }
  end

  def set_locale
    locale_requested = params[:locale]
    I18n.locale = locale_requested and return if locale_requested && locale_reque\
sted.to_sym.in?(I18n.available_locales)

    locale_requested = accept_language_locale
    I18n.locale = locale_requested and return if locale_requested && locale_reque\
sted.to_sym.in?(I18n.available_locales)

    I18n.locale = I18n.default_locale
  end

  def accept_language_locale
    request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first
  end
end
```

(the above code snippet online)

Look at lines 31 to 43. We have adapt `set_locale` to check (line 35 and 36) for the `Accept-Language` header. The juice of the implementation is the method `accept_language_locale` which scans the header value in order to get the first 2 letters. These letters are then used to set the locale, if the locale is supported.

Now, let's try that. In the following video, we demonstrate how the `Accept-Language` header affects the locale that our Web application is using. You have to visit the http://localhost:3000/articles page and also be able to change your Google Chrome language settings. When you set your preferred language to Spanish, then you will see the list of Articles page displaying the Spanish header.

Locale Is Set Based on Accept Language Header Value

Get the Locale From User Preferences

And now, let's implement the last option for our `blog` application to get the locale from. It will be a preference of the user.

Add `locale` to Users

Let's add the column `locale` to `users` table. We first generate the migration:

```
blog $ bin/rails generate migration add_locale_to_users
      invoke  active_record
      create    db/migrate/20170726050032_add_locale_to_users.rb
blog $
```

(the above code snippet online)

Edit the file `db/migrate/20170726050032_add_locale_to_users.rb` and set the following content:

```ruby
# File: db/migrate/20170726050032_add_locale_to_users.rb
#
class AddLocaleToUsers < ActiveRecord::Migration[5.1]
  def change
    add_column :users, :locale, :string, limit: 2, default: 'en', null: false
  end
end
```

(the above code snippet online)

Now, run the migration:

```
blog $ bin/rake db:migrate
== 20170726050032 AddLocaleToUsers: migrating ================================
-- add_column(:users, :locale, :string, {:limit=>2, :default=>"en", :null=>false})
   -> 0.0447s
== 20170726050032 AddLocaleToUsers: migrated (0.0448s) =======================

blog $
```

(the above code snippet online)

Then you can check the status of the `users` table:

```
blog $ psql -d blog_development -c "\d+ users"
                                                                Table "public.use\
rs"
          Column          |            Type             |                     Mod\
ifiers                      | Storage  | Stats target | Description
--------------------------+-----------------------------+------------------------\
----------------------------+----------+--------------+-------------
 id                       | bigint                      | not null default nextva\
l('users_id_seq'::regclass) | plain    |              |
 email                    | character varying           | not null               \
                            | extended |              |
 password_digest          | character varying           | not null               \
                            | extended |              |
```

```
 created_at                | timestamp without time zone | not null                \
                           | plain    |                  |
 updated_at                | timestamp without time zone | not null                \
                           | plain    |                  |
 email_confirmation_token | character varying           |                         \
                           | extended |                  |
 terms_of_service          | boolean                     | not null default false \
                           | plain    |                  |
 country_id                | bigint                      |                         \
                           | plain    |                  |
 locale                    | character varying(2)        | not null default 'en'::\
character varying          | extended |                  |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_email_uidx" UNIQUE, btree (email)
    "users_country_idx" btree (country_id)
Foreign-key constraints:
    "users_country_fk" FOREIGN KEY (country_id) REFERENCES countries(id)
Referenced by:
    TABLE "articles" CONSTRAINT "articles_user_id_users_id_fk" FOREIGN KEY (user_\
id) REFERENCES users(id)

blog $
```

(the above code snippet online)

Do you see the last column added?

Also, you can check that your users now have a default locale set to `en`:

```
blog $ psql -d blog_development -c "select id, email, locale from users"
 id |         email         | locale
----+-----------------------+--------
 10 | mary@gmail.com        | en
 12 | john@gmail.com        | en
 16 | paul@gmail.com        | en
 17 | kristin@gmail.com     | en
 18 | angela@gmail.com      | en
 19 | peter2@gmail.com      | en
 20 | john2@gmail.com       | en
 21 | kelly@gmail.com       | en
 24 | foo-bar@gmail.com     | en
 26 | paulmc@mailinator.com | en
 27 | panos21@gmail.com     | en
 28 | panos22@gmail.com     | en
 29 | panos23@gmail.com     | en
 30 | panos24@gmail.com     | en
```

```
(14 rows)

blog $
```

(the above code snippet online)

Cool!

Update `User` Model With `locale`

Let's update our `User` model with the `locale` attribute validation.

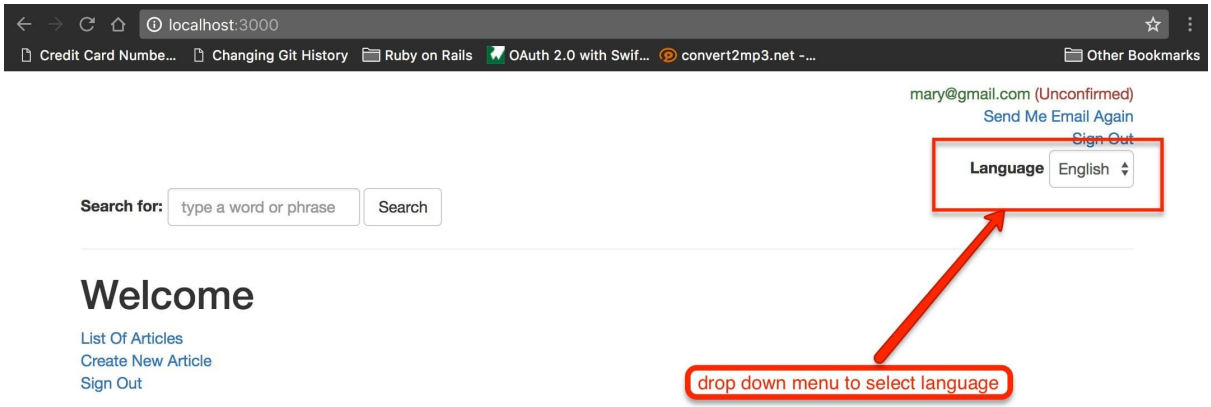
```
1  # File app/models/user.rb
2  #
3  class User < ApplicationRecord
4  ...
5    validates :locale, inclusion: {in: I18n.available_locales.map(&:to_s) }
6
7  ...
8  end
```

(the above code snippet online)

Do you see line 12? This is the validation that we have added for the `locale` attribute of the `User`. Hence, we make sure that the locale set for a `User` is among the available ones.

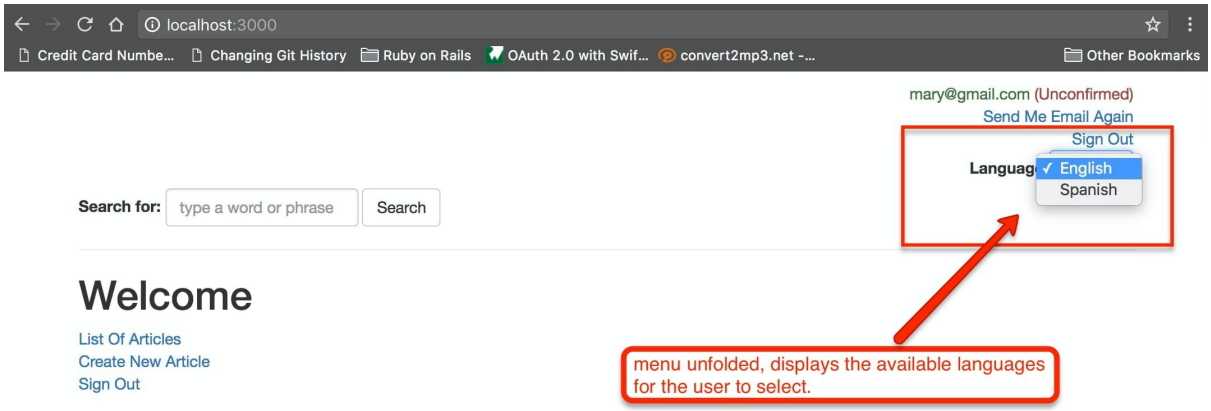
Allow User to Set Their Locale

For the logged in users, we will be offering them a drop down menu to allow them to change their language/locale. Something like this:



Drop down Menu to Select Language

And this is the screenshot when the menu is expanded:



Menu With Languages Expanded

The UI To Select Language

We will incorporate the drop down select box inside the application layout. Like this:

```
1  - # File: app/views/layouts/application.html.haml
2  - #
3  !!!
4  %html
5  ...
6        = render partial: 'layouts/signed_in_user'
7        = render partial: 'layouts/select_language'
8        = render partial: 'layouts/search'
9  ...
10       = yield
```

(the above code snippet online)

Do you see line 15? This is the new line that we have added. It is rendering the partial `app/views/layouts/_select_language.html.haml`. Let's create this new partial file:

```
- # File: app/views/layouts/_select_language.html.haml
- #
- if current_user
  .text-right
    = form_tag '/users/locale', class: 'form-inline' do
      = label_tag :locale, 'Language'
      = select_tag :locale, options_for_select([['English', 'en'], ['Spanish', 'e\
s']]), class: 'form-control'
```

(the above code snippet online)

These are the things that you need to take care of:

1. There is a form that will submit its data to `/users/locale` endpoint. We don't have a route for this. We will have to create it inside the `config/routes.rb`.
2. There is a select box that we build with the `select_tag` helper. Also we use the `options_for_select` to build the `<option>` tags. What we give there is an Array of Arrays. Each element is an Array that has two elements, the Language text and the Language locale symbol.
3. There is no submit button to submit the form. We do not want to ask the user to click on a button in order to change their language. We want this to be fired as soon as the user changes the selection in the select box. In order to achieve this, we will hook a JavaScript code at the select box that would submit the form when select box would change its value.
4. The above HAML for selecting the language is not the final one and you will see that we will have to do some enhancements.

If you save the above and you load a page as logged in user in your `blog` application, you will see the select box with the options. Of course, when changing, nothing would take place for the time being.

POST the Form

In order to POST the form to the server, we will need to hook JavaScript code to the select box. This is how we do that. We create the file `app/assets/javascripts/select_language.js` with the following code:

```
// File: app/assets/javascripts/select_language.js
//
$(document).ready(function () {
    $('select#locale').on('change', function () {
        var $selectLanguageForm = $(this).closest('form');
        $selectLanguageForm.submit();
    });
});
```

(the above code snippet online)

The code is very easy to understand, since you have already done the jQuery chapter. It attaches a handler to the `change` event of the select box (which is selected with `$('select#locale')`. The handler locates the form using the jQuery method `closest()`. If you want to read about `closest()`, you can read here. Then it calls the jQuery method `submit()` on the form located, which submits the form.

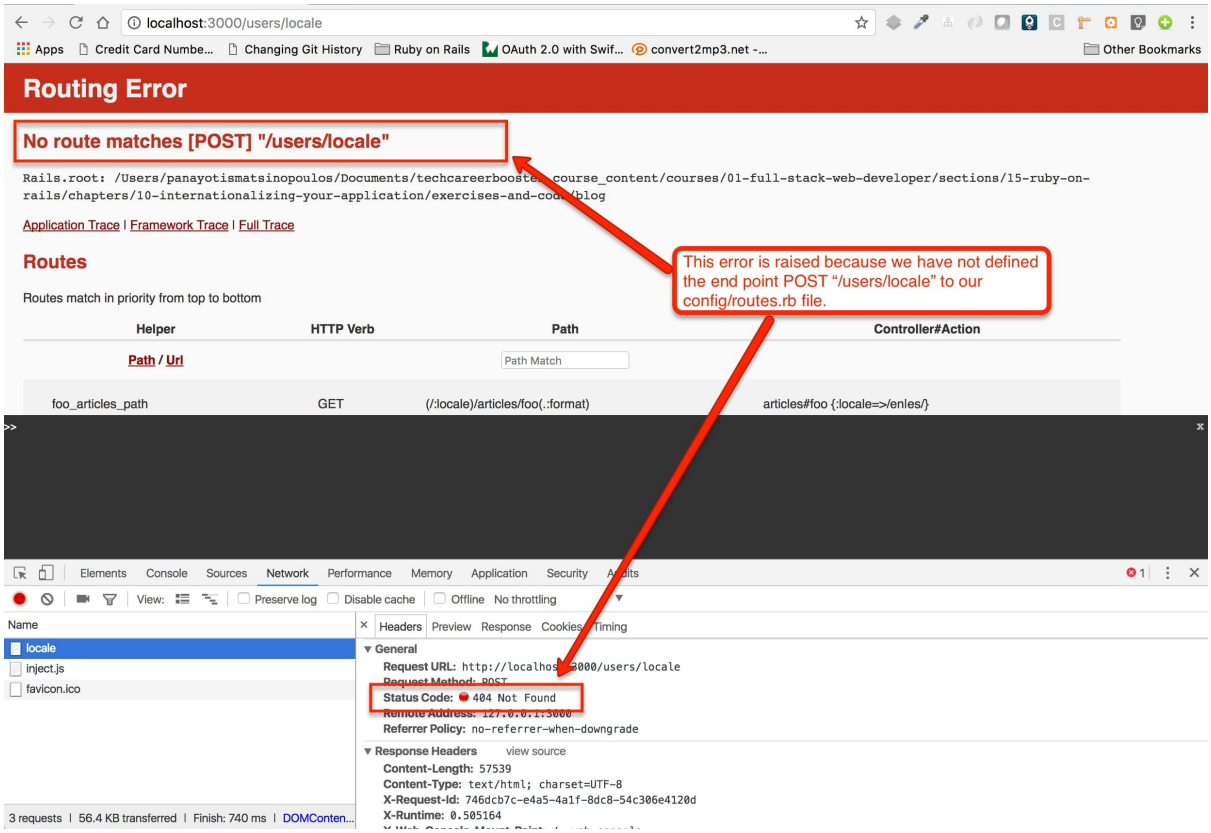
With this code in place, we don't need a submit button. The form is submitted as soon as the user changes the value in the select box with the languages.

> *Hint:* If you wanted to find the id of the select box in order to specify the proper selector (`select#locale` because id is `locale`), and since this information is not very obvious from the HAML code, you can use the Google Chrome developer tools to inspect the select box and find out the id. Nevertheless, you need to know that the `select_tag` creates select box with id equal to the first argument, if `id` is not explicitly specified.

> **Important** Rails comes with the turbolinks gem installed. However, this gem might conflict with jQuery because it affects how links work. Our suggestion is that you remove the turbolinks from your project. Here is what you have to do:

> 1. remove the gem 'turbolinks' from your Gemfile and do `bundle`.
> 2. remove line `require turbolinks` from your `app/assets/javascripts/application.js` file.
> 3. remove `'data-turbolinks-track': 'reload'` from the `app/views/layouts/application.html.haml`, line 9 and 10.
> 4. restart your server

Now, if you save your code, reload your page and try to change the value of the language from English to Spanish, you will get the following error:



POST Users Fails Due To Missing Route

Route and Handling Action

Now, we need to specify the endpoint `POST /users/locale`. Let's do that in the `config/routes.rb` file:

```
1  # File: config/routes.rb
2  #
3  Rails.application.routes.draw do
4    post '/users/locale', to: 'users#update_locale'
5
6    scope "(:locale)", locale: Regexp.new(I18n.available_locales.map(&:to_s).join('\
7  |')) do
8      resources :articles do
9        collection do
10 ...
11     root 'welcome#index'
12   end
13 end
```

(the above code snippet online)

The new line is line 4. Note that we didn't put this new route inside the `:locale` scope. This is because we do not want the path to have `:locale` part as an optional part.

Run the `bin/rake routes` command to see which route this new entry defines:

```
blog $ bin/rake routes
                      Prefix Verb   URI Pattern                                \
     Controller#Action
                users_locale POST   /users/locale(.:format)                    \
     users#update_locale
                foo_articles GET    (/:locale)/articles/foo(.:format)          \
     articles#foo {:locale=>/en|es/}
                bar_articles GET    (/:locale)/articles/bar(.:format)          \
     articles#bar {:locale=>/en|es/}
...
blog $
```

(the above code snippet online)

Do you see first line? This is the line added by the new routes entry. With this entry in place we have the method `users_locale_path` at our disposal. Let's use it inside our `app/views/layouts/_select_language.html.haml` partial:

So, instead of `form_tag '/users/locale'` go ahead and use `form users_locale_path(locale: nil)` instead. Note that the `locale: nil` is needed, otherwise, the `default_url_options` will build the URL with the `locale` in place, which is not what you want for this particular case.

Now that the routes and view are ready, try to change the language to Spanish again. You will get the error that `UsersController` is an uninitialized constant. That was expected actually. The route `users#update_locale` means that we need to have an action `update_locale` inside a controller named `UsersController`. Let's implement this:

```ruby
# File: app/controllers/users_controller.rb
#
class UsersController < ApplicationController
  skip_before_action :set_locale, only: [:update_locale]

  def update_locale
    locale_requested = params[:locale]
    if locale_requested.present? && locale_requested.to_sym.in?(I18n.available_lo\
cales)
      current_user.update_attribute(:locale, locale_requested)
    end

    referer = request.headers['Referer']
    if referer
      url_to_redirect_to = Rails.application.routes.recognize_path(referer).merge\
({locale: params[:locale]})
    else
      url_to_redirect_to = root_url(locale: params[:locale])
    end

    redirect_to url_to_redirect_to
```

```
22    end
23  end
```

(the above code snippet online)

Things that you need to make sure you take a notice of:

1. We skip the setting of locale on line 4. This is because when updating the locale of the user we then want to just redirect. The redirect (line 19) will take care of resubmitting a new request which is going to set the locale anyway.
2. We redirect to either the `root_url` or to the page the user came from.
3. The page the user came from is inside the `Referer` header of the request. If it is present, then we build a new URL based on the value of this header. The new URL is the same as the URL in the referer, except from the fact that we replace the `locale` with the locale specified in the params. Hence, if the user is coming from `http://localhost:3000/en/articles` and the `update_locale` is called to update the locale to Spanish, then we redirect back to `http://localhost:3000/es/articles`. In order to achieve this, we decompose the referer value to its parts using the `Rails.application.routes.recognize_path` method. Look how this method works by trying it out in a rails console:

```
2.3.3 :001 > Rails.application.routes.recognize_path('http://localhost:3000/en/ar\
ticles')
 => {:controller=>"articles", :action=>"index", :locale=>"en"}
```

4. If we don't have a value from the `Referer` header, we use the `root_url`, but we make sure that the locale we set there is the locale selected in params.

Update Logic on Set Locale

Still, we need to update the logic in the `set_locale` method to take into account the current user preferences:

```
# File: app/controllers/application_controller.rb
#
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
...
  def set_locale
    I18n.locale = current_user.locale and return if current_user

    locale_requested = params[:locale]
    I18n.locale = locale_requested and return if locale_requested && locale_reque\
sted.to_sym.in?(I18n.available_locales)

    locale_requested = accept_language_locale
    I18n.locale = locale_requested and return if locale_requested && locale_reque\
sted.to_sym.in?(I18n.available_locales)
```

```

    I18n.locale = I18n.default_locale
  end
...
end
```

(the above code snippet online)

The logic has changed so that we take into account, first the preferences of the current user. Otherwise, we use the locale defined in params or the locale defined in `Accept-Language` header. The last resort is the `I18n.default_locale`

> *Hint:* Did you notice the `and return` usage? It allows us to do things and exit method in one liners.

Language Select Box To Display Users Locale

What we have to do though, is to make sure that the language select box is selecting the current user locale:

```
1 - # File: app/views/layouts/_select_language.html.haml
2 - #
3 - if current_user
4   .text-right
5     = form_tag '/users/locale', class: 'form-inline' do
6       = label_tag :locale, 'Language'
7       = select_tag :locale, options_for_select([['English', 'en'], ['Spanish', 'e\
8 s']], current_user.locale), class: 'form-control'
```

(the above code snippet online)

Do you see line 7? We have updated the `options_for_select` with an extra, last, argument: `current_user.locale`.

Now everything is ready. You can see how the user can use the language select box to change their language preference. Here is the video that demonstrates that:

Locale Is Set Based on Accept Language Header Value

Passing Variables To Translations

The strings stored inside your translation database, i.e. inside your `config/locales/*.yml` files can be set up so that they use variable parts. For example, let's work out with the string in the success flash message when an Article is destroyed:

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
...
  end

  def destroy
    @article.destroy
    flash[:success] = 'Article has been destroyed!'
    redirect_to articles_path
  end

  def foo
...
end
```

(the above code snippet online)

Do you see line 43? `flash[:success] = 'Article has been destroyed!'`. The string `Article has been destroyed!` is hard coded in English. Also, we may want to tell the user which article (by its title) has been destroyed, because after the destroy user lands on the list of articles page. Telling user the Article title that has been destroyed would have been for better User experience.

We can do that by changing the code as follows:

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
...
  end

  def destroy
    @article.destroy
    flash[:success] = I18n.t('article_has_been_destroyed', article_title: @articl\
e.title)
    redirect_to articles_path
  end

  def foo
...
end
```

(the above code snippet online)

We have replaced the line 43 with this: `flash[:success] = I18n.t('article_has_been_destroyed', article_title: @article.title)`.

The `I18n.t` call takes as first argument the key to the translation database and a Hash with the variable parts. In this particular case we only have 1 variable part with key `article_title` the

value of which we set it to the `@article.title`, i.e. to the title of the article that has just been deleted.

Then, we need to make sure that we have the correct keys inside our `en.yml` and `es.yml` files:

```
# File: config/locales/en.yml
#
en:
  hello: "Hello World"
  list_of_articles: "List of Articles"
  article_has_been_destroyed: "Article has been destroyed! Article Title: %{artic\
le_title}"
```

(the above code snippet online)

and

```
# File: config/locales/es.yml
#
es:
 hello: "Hola Mundo"
 list_of_articles: "Lista de artículos"
 article_has_been_destroyed: "El artículo ha sido destruido! Título del artículo:\
 %{article_title}"
```

(the above code snippet online)

Do you see how we define the variable parts of the translation key value? See lines 6 in both files. The variable part is enclosed inside `%{ ..... }` and it includes a Hash key name, the key name that needs to be used inside the code in order to provide the actual value of the variable part.

Matching Variable Parts of a Translation

With these things in place, then try to delete an article, both while with English and while with Spanish:

Locale Is Set Based on Accept Language Header Value

You can see how the variable part of the translated string is being dealt with.

Displaying Date/Time

Displaying date/time on the views is affected by the locale selected. For example, other languages prefer the format `DD/MM/YYYY` whereas others, they prefer the format `MM/DD/YYYY`, or `YYYY/MM/DD`.

Let's see how we can localize the display of timestamps:

Add Timestamp To Article Index Page

In order to see how we should be displaying dates and times, let's add a new column to the Articles index page that would display the dates the Articles were created.

```
- # File: app/views/articles/index.html.haml
- #
%h1= t 'list_of_articles'
...
      %th Created At
      %th
...
    = link_to 'Home', root_path
```

(the above code snippet online)

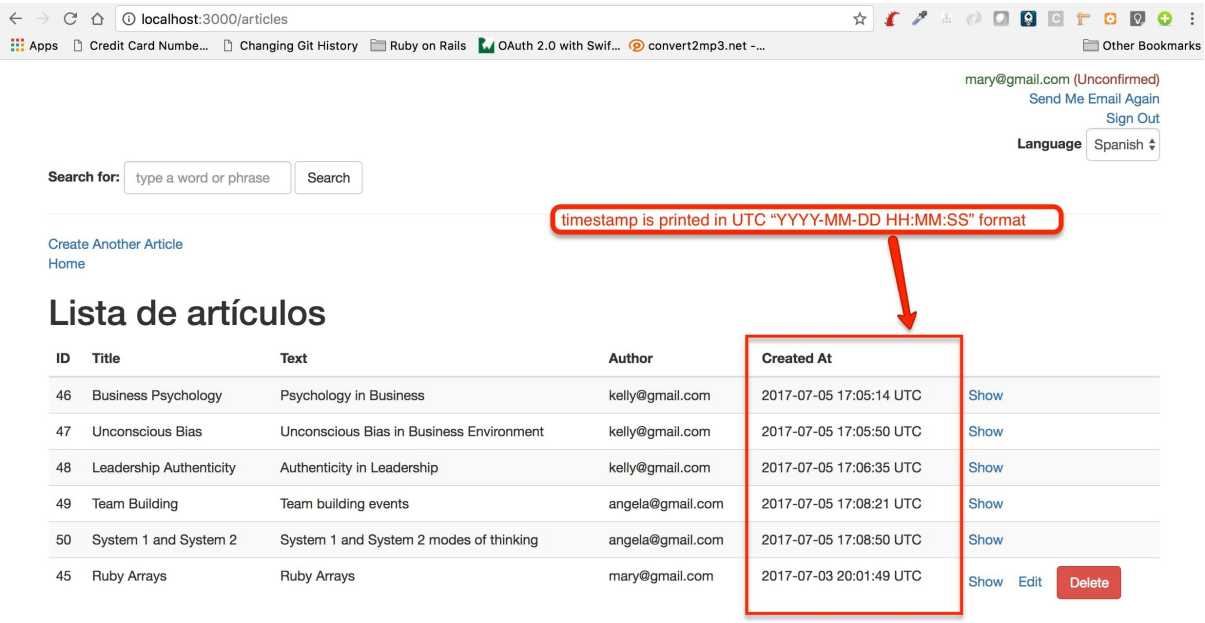
Inside the `app/views/articles/index.html.haml` we have added line 11, as the column header. Then we need to amend the partial `app/views/articles/_article.html.haml` which is the partial used to render each article in the list:

```
1 - # File: app/views/articles/_article.html.haml
2 - #
3 ...
4   %td= article.created_at
5   %td
6 ...
7       = link_to 'Delete', article_path(article), method: :delete, class: 'btn btn\
8 -danger', data: {confirm: 'Are you sure that you want to delete this Article?'}
```

(the above code snippet online)

We have added line 8, that displays the value of `created_at` attribute of the Article.

If you visit the http://localhost:3000/articles page you will see something like this:



Standard Format in UTC Timestamp

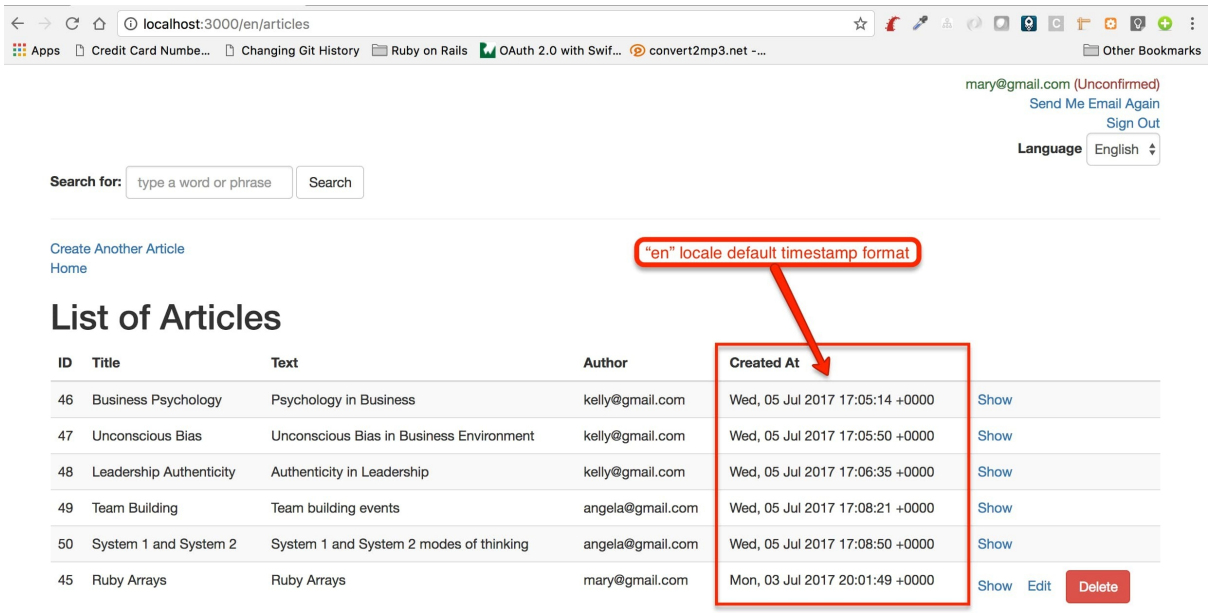
You can see that, by default, the timestamps are not very user-friendly. Firstly, they display the time in UTC and not in the timezone of the user. But this timezone problem is going to be covered later on. Secondly, the format is quite *ugly*: YYYY-MM-DD HH:MM:SS.

Let's use the `I18n.locale` method (just `l` if used in views) in order to display the date/time:

```
- # File: app/views/articles/_article.html.haml
- #
...
  %td= l article.created_at
  %td
...
      = link_to 'Delete', article_path(article), method: :delete, class: 'btn btn\
-danger', data: {confirm: 'Are you sure that you want to delete this Article?'}
```

(the above code snippet online)

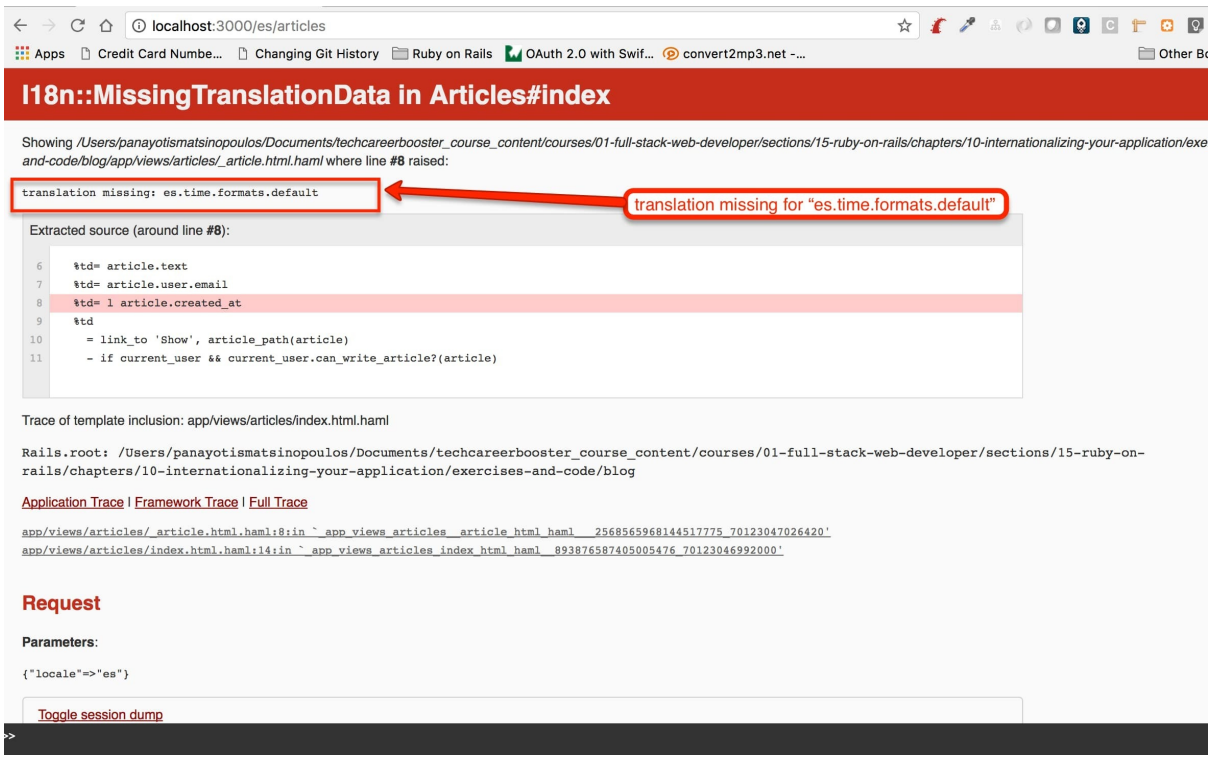
And lets reload the page http://localhost:3000/articles, but try first with the english version:



The Default Format for Timestamps When en Locale

As you can see, the default format of timestamps when `en` is the locale, is much better than the previous one.

But, if you try to get the same page using the Spanish language, you will get an error:



Missing Translation for Default Time Format in Spanish

As you can read from the error message, our translation database is missing the Spanish translation for the default time format. And watch for the actual key reported as missing: `es.time.formats.default`. It is the a hierarchical key. We are missing `default`, which should be in the list of keys of `formats`, which, in turn, should be in the list of keys in `time`, which, in turn, should be in the list of keys in `es`.

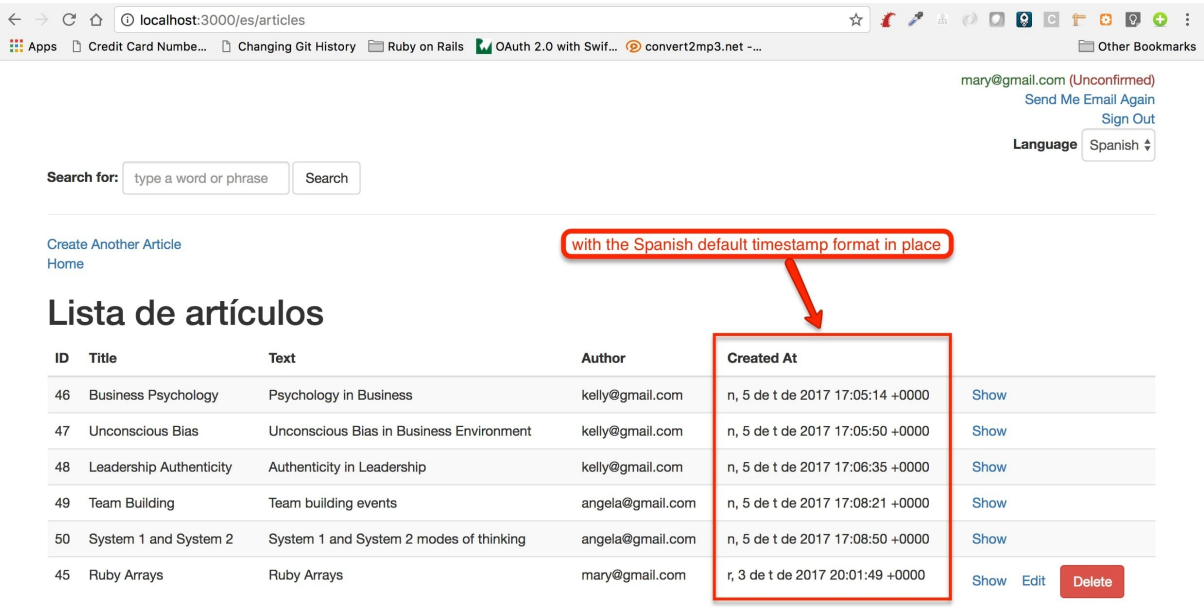
Update your `es.yml` file and make it look as follows:

```
# File: config/locales/es.yml
#
es:
  hello: "Hola Mundo"
  list_of_articles: "Lista de artículos"
  article_has_been_destroyed: "El artículo ha sido destruido! Título del artículo\
: %{article_title}"
  time:
    formats:
      default: "%A, %-d de %B de %Y %H:%M:%S %z"
```

(the above code snippet online)

Do you see how we have added a new entry for `es.time.formats.default` ? It has the value `"%A, %-d de %B de %Y %H:%M:%S %z"` and it is a format specification for string representation of timestamps. The symbols that exist inside it, can be explained if one reads the documentation here.

With this update in place, let's try to reload the Spanish version of the list of articles again:



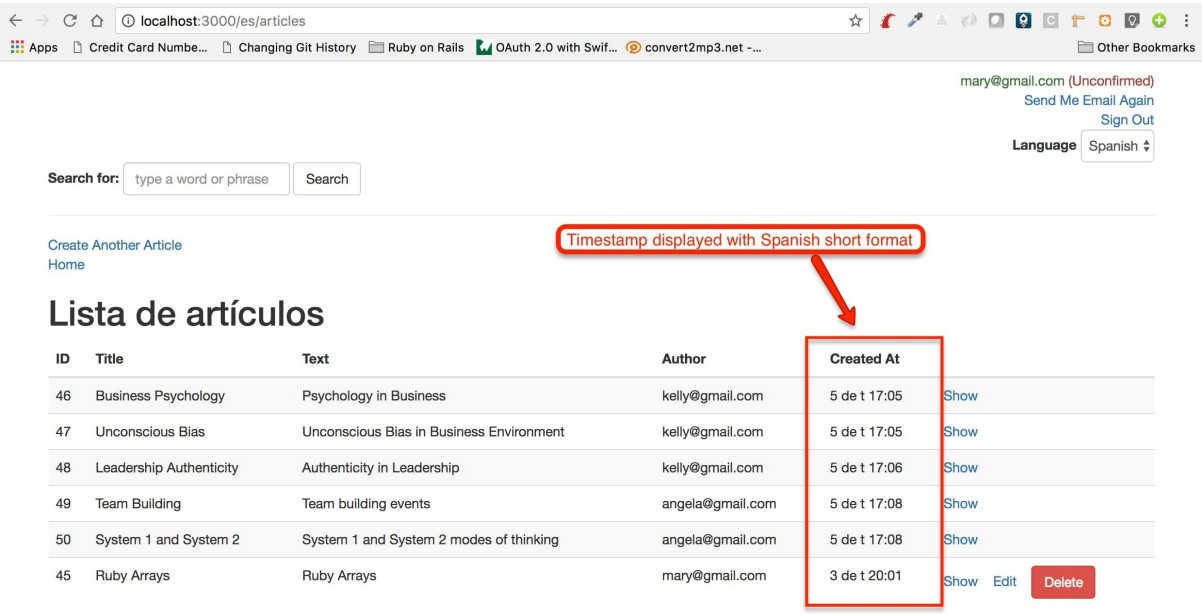
Default Format for Spanish Timestamp

But again, the `default` is not the only format that you can use. You can define and use your own formats of course. Let's create the `short` format and tell our page to use this:

```
# File: config/locales/es.yml
#
es:
  hello: "Hola Mundo"
  list_of_articles: "Lista de artículos"
  article_has_been_destroyed: "El artículo ha sido destruido! Título del artículo\
: %{article_title}"
  time:
    formats:
      default: "%A, %-d de %B de %Y %H:%M:%S %z"
      short: "%-d de %b %H:%M"
```

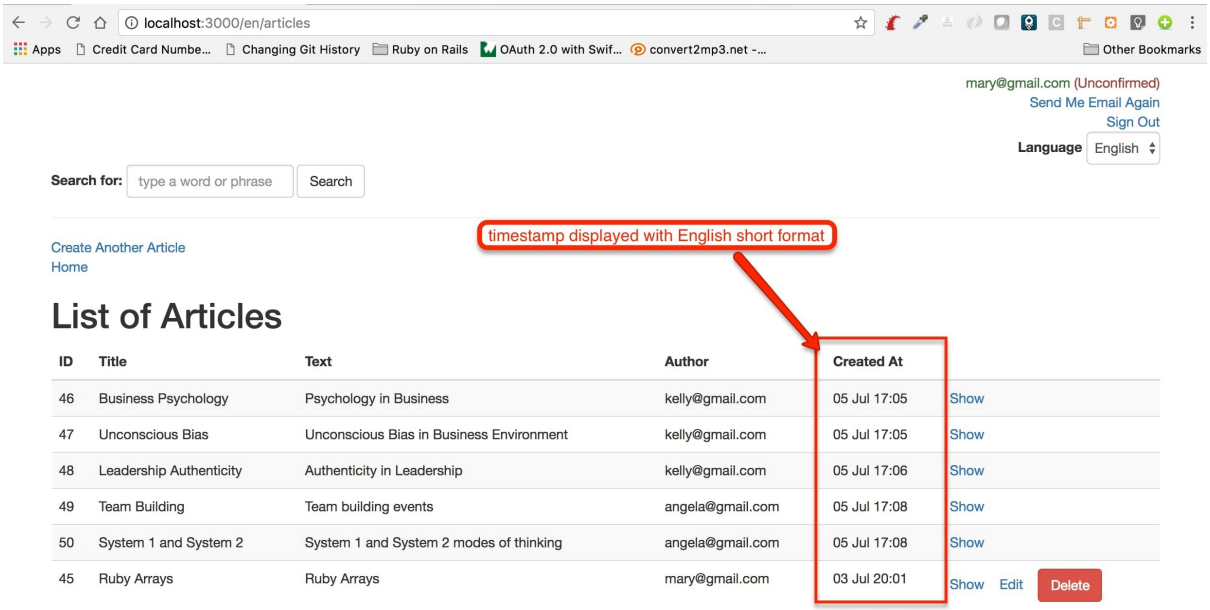
(the above code snippet online)

In order to tell our view to print the timestamp in short format, instead of just calling `l article.created_at` we call `l article.created_at, format: :short`. And this is what we get for the Spanish page:



Timestamp Displayed with Spanish Short Format

But it works for English too:



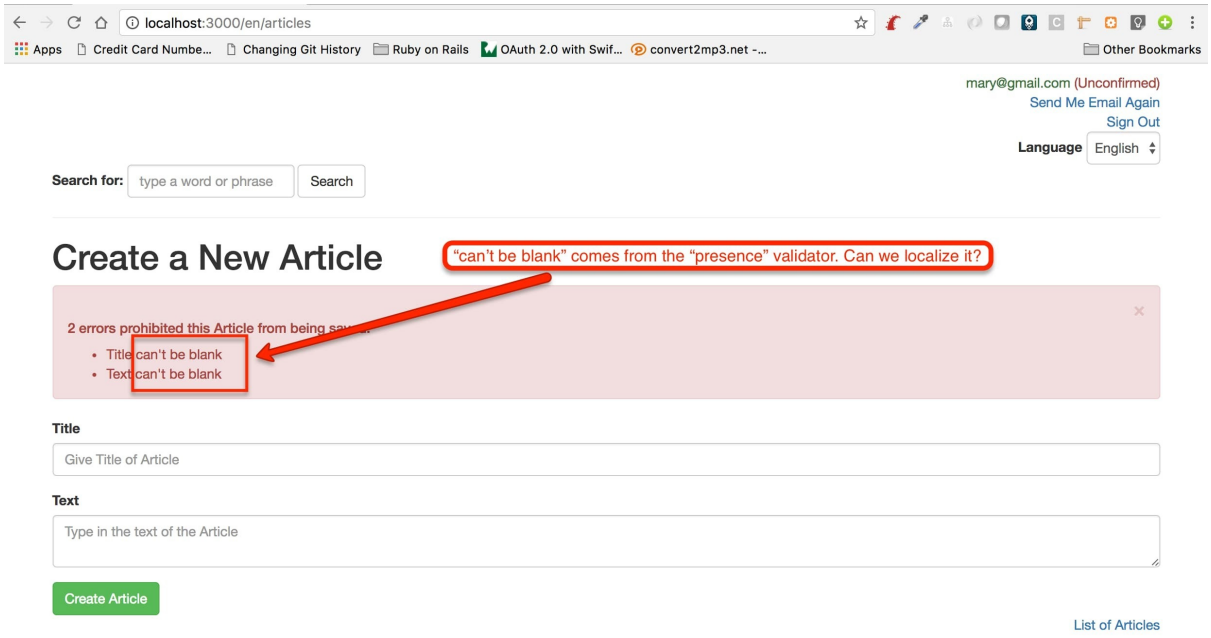
Timestamp Displayed With Short English Format

How does it work, by the way? We have not defined the `:short` format for the English locale. Neither the `:default` one. It works because Rails comes ready with all the translations for English. This is actually part of the ActiveSupport.

Now, if you want to have translations for your language for all the Rails internal things, like the formatting of timestamps, you may want to pick up the `locales/*.yml` file that you are interested in from this rails-i18n repo, since it seems that somebody else has already done a lot of the translation work for many of the languages and locales out there.

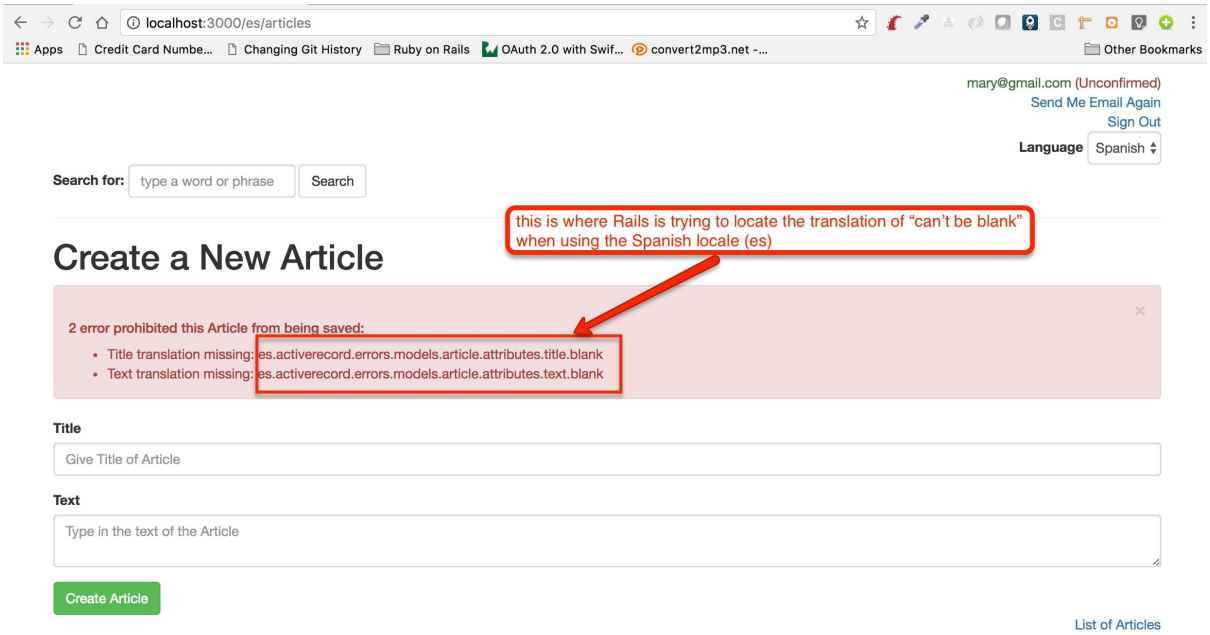
Validation Error Messages

Now, let's see how we can localize the validation error messages. Look for example what it is displayed when you try to create an article without filling in the form with correct data:



Validation Messages Need Localization

If you try the same function using Spanish locale, you will also see where Rails is trying to locate the translations of the `"can't be blank"` string:



Where Rails Is Trying to Locate Translation for Cant Be Blank In Spanish

As you can see from the error raised, Rails is trying to locate the translation for `"can't be blank"` to `es.activerecord.errors.models.article.attributes.title.blank`.

In fact, Rails is looking in the following locations for the translations of the key `blank`:

```
es.activerecord.errors.models.[model_name].attributes.[attribute_name].blank
es.activerecord.errors.models.[model_name].blank
es.activerecord.errors.messages.blank
es.errors.attributes.[attribute_name].blank
es.errors.messages.blank
```

(the above code snippet online)

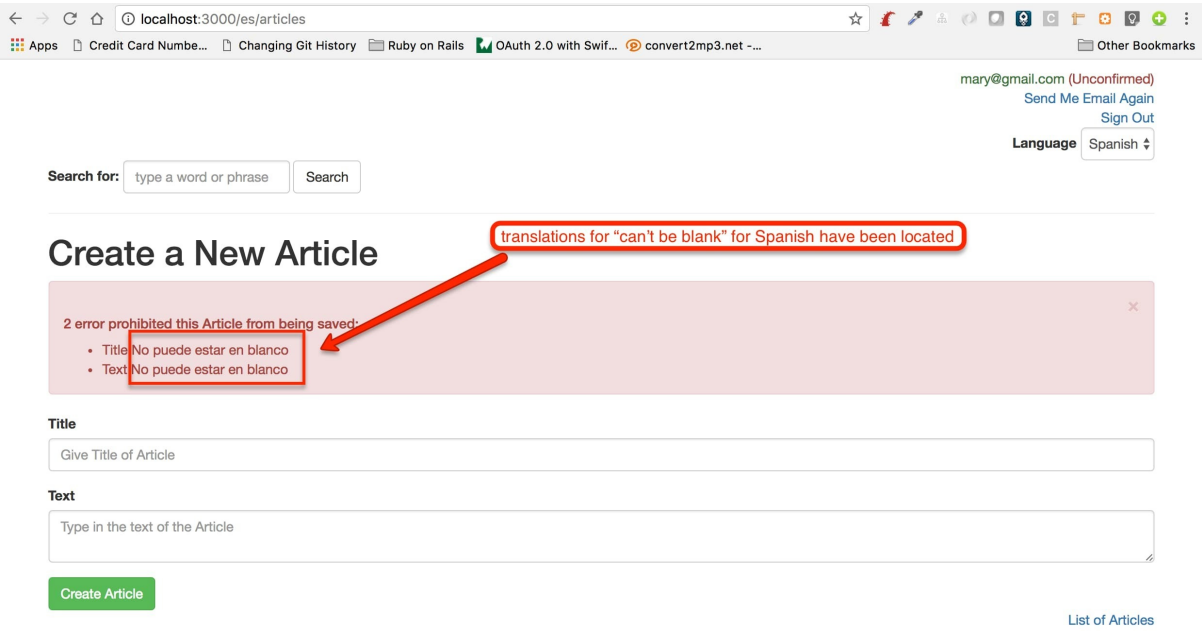
So, let's update our `config/locales/es.yml` file accordingly, by going with the last option from the above list:

```
# File: config/locales/es.yml
#
es:
  hello: "Hola Mundo"
  list_of_articles: "Lista de artículos"
  article_has_been_destroyed: "El artículo ha sido destruido! Título del artículo\
: %{article_title}"
  time:
    formats:
      default: "%A, %-d de %B de %Y %H:%M:%S %z"
      short: "%-d de %b %H:%M"
```

```
12    errors:
13      messages:
14        blank: "No puede estar en blanco"
```

(the above code snippet online)

Do you see lines 11 to 13? These are there to translate `"can't be blank"` in Spanish. Now try to create an article without giving any input and using the Spanish locale. You will see this:



Translations For Cant Be Blank in Spanish Have Been Located

User Time Zone

Since your Web application might be used by users all over the world, we need to take into account the User time zone when we display timestamps at the user interface.

List All Time Zones

You can get the list of time zones that Rails supports by running the rake task `"time:zones"`:

(You may want to use `bin/rake time:zones | more` to be able to view the time zones with pagination on your terminal output)

```
blog $ bin/rake time:zones

* UTC -11:00 *
American Samoa
International Date Line West
Midway Island

* UTC -10:00 *
Hawaii

* UTC -09:00 *
Alaska

* UTC -08:00 *
Pacific Time (US & Canada)
Tijuana

...


* UTC +12:00 *
Auckland
Fiji
Kamchatka
Marshall Is.
Wellington

* UTC +12:45 *
Chatham Is.

* UTC +13:00 *
Nuku'alofa
Samoa
Tokelau Is.
```

(the above code snippet online)

The above lists the UTC offsets and the names of the time zones that fall at this particular offset.

You can also get the list of time zones per country too. Use the two digit country code. For example, for Germany:

```
blog $ bin/rake time:zones[DE]


* UTC +01:00 *
Berlin
Europe/Busingen

blog $
```

(the above code snippet online)

you get 1 offset and two time zone names: 1) `Berlin` and 2) `Europe/Busingen`

Set the Application Default Time Zone

You can set the default time zone of the application inside the `config/application.rb`. Let's suppose that we want the default time zone to be `Eastern Time (US & Canada)`. Let's do that:

```
1  # File: config/application.rb
2  #
3  require_relative 'boot'
4
5  require 'rails/all'
6
7  # Require the gems listed in Gemfile, including any gems
8  # you've limited to :test, :development, or :production.
9  Bundler.require(*Rails.groups)
10
11 module Blog
12   class Application < Rails::Application
13 ...
14
15     config.time_zone = "Eastern Time (US & Canada)"
16   end
17 end
```

(the above code snippet online)

Do you see line 23? This is how we set the default time zone to `Eastern Time (US & Canada)`.

Let's prove that this one works by running the following:

```
blog $ bin/rails runner "puts Time.zone.now"
2017-07-27 13:52:19 -0400
blog $ bin/rails runner "puts Time.zone"
(GMT-05:00) Eastern Time (US & Canada)
```

(the above code snippet online)

The `Time.zone.now` prints the current time in the current time zone. Also the `Time.zone` prints the current time zone.

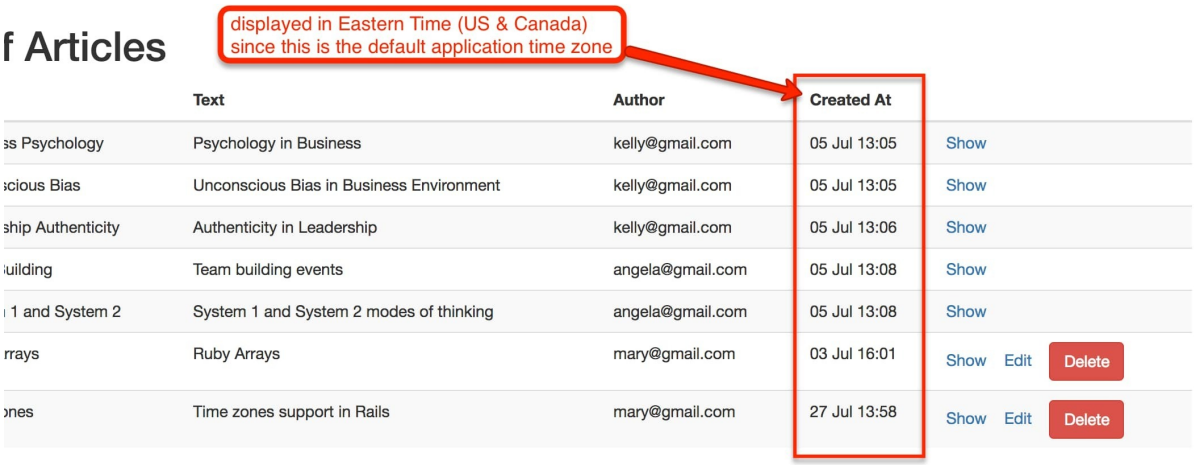
Also, the `Time.zone.now.utc` will convert the time to UTC:

```
blog $ bin/rails runner "puts Time.zone.now.utc"
2017-07-27 17:57:18 UTC
blog $
```

(the above code snippet online)

Timestamps

Note that the timestamps displayed in Rails are automatically converted to the current time zone:



Timestamps Displayed In Current Time Zone

User Time Zone

But, when a user is logged in, they usually want to see the timestamps in their own time zone, not the default time zone of the application.

First, we need to allow user to select their time zone. In order to do that, like we did for locales, we need to introduce a new column in `users` table. This new column is going to be called `time_zone`.

`time_zone` column

Let's generate the migration file.

```
blog $ bin/rails generate migration add_time_zone_to_users
      invoke  active_record
      create    db/migrate/20170727192758_add_time_zone_to_users.rb
blog $
```

(the above code snippet online)

Let's edit the file `db/migrate/20170727192758_add_time_zone_to_users.rb` so that it has the content:

```
# File: db/migrate/20170727192758_add_time_zone_to_users.rb
#
class AddTimeZoneToUsers < ActiveRecord::Migration[5.1]
  def change
    add_column :users, :time_zone, :string, default: Rails.application.config.tim\
e_zone, null: false
  end
end
```

(the above code snippet online)

According to the above, we will add the column `time_zone` as a string column with default value equal to the default time zone of the application. This is a mandatory attribute for all users.

Let's run the migration:

```
blog $ bin/rake db:migrate
== 20170727192758 AddTimeZoneToUsers: migrating ================================
-- add_column(:users, :time_zone, :string, {:default=>"London", :null=>false})
   -> 0.0490s
== 20170727192758 AddTimeZoneToUsers: migrated (0.0492s) =======================

blog $
```

(the above code snippet online)

And confirm that the users now have time zone:

```
blog $ psql -d blog_development -c "select id, email, time_zone from users limit \
5;"
 id |       email       |         time_zone
----+-------------------+---------------------------
 12 | john@gmail.com    | Eastern Time (US & Canada)
 16 | paul@gmail.com    | Eastern Time (US & Canada)
 17 | kristin@gmail.com | Eastern Time (US & Canada)
 18 | angela@gmail.com  | Eastern Time (US & Canada)
 19 | peter2@gmail.com  | Eastern Time (US & Canada)
(5 rows)
blog $
```

(the above code snippet online)

content is owned by Tech Career Booster (https://www.techcareerbooster.com)

`User` Model

Let's enhance our `User` model to make sure that we don't create Users without their timezone:

```
# File app/models/user.rb
#
class User < ApplicationRecord
  DEFAULT_TIME_ZONE = Rails.application.config.time_zone
...
  validates :time_zone, inclusion: { in: ActiveSupport::TimeZone.all.map(&:name) }

  before_validation :set_time_zone
...
  def set_time_zone
    self.time_zone = DEFAULT_TIME_ZONE unless time_zone.present?
  end
end
```

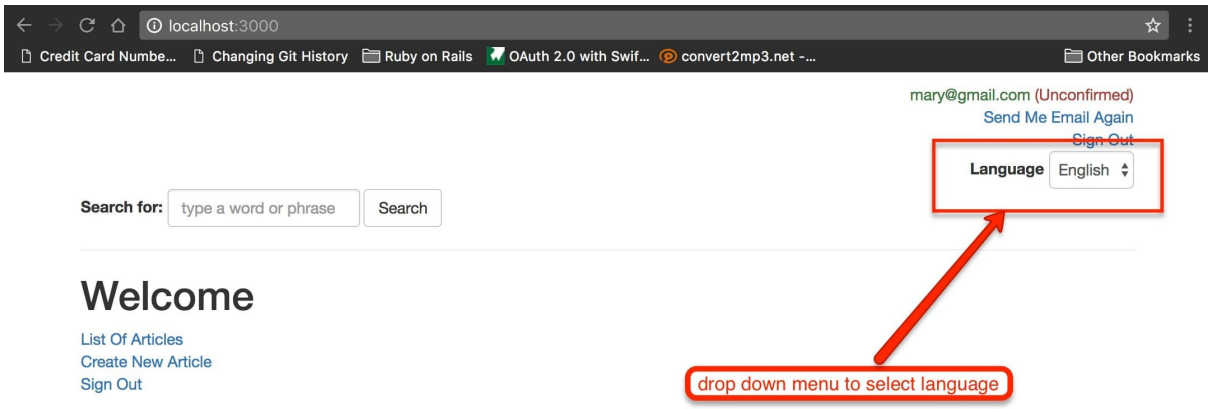
(the above code snippet online)

1. We declare the default time zone to be the default time zone of the application
2. We also use a `before_validation` to make sure that we set the default value to whoever User instance does not have one time zone set.

Now, whenever a new user signs up, their time zone will be set to the default time zone of the application.

Allow User To Change Time Zone

Like we allow user to change locale, we will allow user to change time zone too. Here is what we want to do:



Dropdown To Allow Users To Select Their Time Zone

The UI

Let's add the HTML markup first. This is the partial that we are going to create `app/views/layouts/_select_time_zone.html.haml`:

```
- # File: app/views/layouts/_select_time_zone.html.haml
- #
- if current_user
  .text-right
    = form_tag '/users/time_zone', class: 'form-inline' do
      = label_tag :time_zone, 'Time Zone'
      = select_tag :time_zone, options_for_select(ActiveSupport::TimeZone.all.map\
(&:name).sort, current_user.time_zone), class: 'form-control'
```

(the above code snippet online)

I guess that this is now pretty much straightforward for you. Note how we build the options for the select. We use the names of the collection `ActiveSupport::TimeZone.all`. Also, we make sure that there is an option selected and this is the time zone of the current user.

This partial needs to be embedded inside the `app/views/layouts/application.html.haml`:

```
1  - # File: app/views/layouts/application.html.haml
2  - #
3  !!!
4  %html
5    %head
6      %meta{:content => "text/html; charset=UTF-8", "http-equiv" => "Content-Type"}/
7      %title Blog
8      = csrf_meta_tags
9      = stylesheet_link_tag    'application', media: 'all'
10     = javascript_include_tag 'application'
11   %body
12     .container
13       = render partial: 'layouts/flash'
14       = render partial: 'layouts/signed_in_user'
15       = render partial: 'layouts/select_language'
16       = render partial: 'layouts/select_time_zone'
17       = render partial: 'layouts/search'
18       = yield :menu
19       = yield
```

(the above code snippet online)

Do you see line 16? This is where we invoke the partial `app/views/layouts/_select_time_zone.html.haml`.

Submit POST On Change

Like we did for the language, we will be submitting the form on change of the select box:

```
1  // File: app/assets/javascripts/select_timezone.js
2  //
3  $(document).ready(function () {
4      $('select#time_zone').on('change', function () {
5          var $selectTimeZoneForm = $(this).closest('form');
6          $selectTimeZoneForm.submit();
7      });
8  });
```

(the above code snippet online)

Define The Route

We define the routes inside the `config/routes.rb`:

```ruby
# File: config/routes.rb
#
Rails.application.routes.draw do
  post '/users/locale', to: 'users#update_locale'
  post '/users/time_zone', to: 'users#update_time_zone'

...
end
```

(the above code snippet online)

We have added the line 5. If you print the routes, you will get this:

```
blog $ bin/rake routes
                      Prefix Verb   URI Pattern                                  \
     Controller#Action
                 users_locale POST   /users/locale(.:format)                      \
     users#update_locale
             users_time_zone POST   /users/time_zone(.:format)                    \
     users#update_time_zone
...
blog $
```

(the above code snippet online)

This means that we can use the path `users_time_zone_path` inside the `app/views/layouts/_select_time_zone.html.haml` file. Let's do the following:

Instead of `form_tag '/users/time_zone'` let's write `form_tag users_time_zone_path`.

Action and Controller Update

Now, we need to update the `UsersController` accordingly:

```ruby
# File: app/controllers/users_controller.rb
#
class UsersController < ApplicationController
  skip_before_action :set_locale, only: [:update_locale, :update_time_zone]
...

  def update_time_zone
    time_zone = params[:time_zone]
    if time_zone.present? && time_zone.in?(ActiveSupport::TimeZone.all.map(&:name\
))
      current_user.update_attribute(:time_zone, time_zone)
    end

    redirect_back(fallback_location: root_url)
```

```
  end
end
```

(the above code snippet online)

We have added the action `update_time_zone`. The code is again pretty straightforward. If a `time_zone` is specified and it is among the valid time zones then we update the time zone of the currently logged in user.

Done! The following video shows how the time zone of the currently logged in user is updated with the drop down changes:

Locale Is Set Based on Accept Language Header Value

However, as you will see from the video and, also, as you will experience too, the timestamps displayed in the List of Articles do not change according to the changes in the User time zone.

This is because the Web application does not take into account the user time zone yet, when displaying timestamps.

Taking User Time Zone Into Account

Taking user time zone into account needs a similar approach to the one for taking user language into account. We need to update `ApplicationController` accordingly:

```
# File: app/controllers/application_controller.rb
#
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  before_action :authenticate_user
  before_action :set_locale
  around_action :set_user_time_zone

  helper_method :current_user
...
  end

  def set_user_time_zone(&block)
    if current_user.nil?
      yield
    else
      Time.use_zone(current_user.time_zone, &block)
    end
  end
end
```

(the above code snippet online)

In order to take into account User time zone, we create an `around_action` filter. The `around_action` filters allow us to execute code before and after the request is being processed.

Here, we call the `set_user_time_zone` method that takes as block the request execution. We use the `Time.use_zone` method to execute the block within the time zone of the user.

Now, every time you change the time zone, the timestamps are updated accordingly on the UI:

Locale Is Set Based on Accept Language Header Value

Closing Note

Ruby on Rails allows you, via the `i18n` gem to use different backends for your translations. But this is out of the scope of this course. Nevertheless, if you want to read more about localizing your Web application, you can read the corresponding Rails Guides here.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz Details

The quiz for this chapter can be found here

Task Details

You need to localize your `blog` application according to the instructions and guidelines presented in the chapter. Don't localize only the parts of the application that are localized within the chapter. Localize everything that you believe should be localized. And use whatever second language (besides English) that you like.

Indicatively but not limited here:

- Make sure you add the correct `.yml` file inside your `config/locales` folder.
- Make sure that you configure your application with a new default locale, if this is not going to be English.
- Make all the text that appears on all the views to be using the `I18n.translate` method. Like we did for `List Of Articles`.
- Make sure that you use a `before_action` filter that would set the locale using a combined logic:
  1. If the user is logged in, then set the locale based on their locale.
  2. If the user is not logged in, but the params hash contains the locale, then use whatever the value is specified.
  3. Otherwise, use whatever is specified in the `Accept-Language` request header.
  4. In any other case, use the default locale specified by the `I18n.default_locale`.

- You need to configure your routes so that the locale can be an optional part of the URL path, and not only part of the query string parameters.
- Make sure that `locale` property on `User` takes values from a limited set of values, by using an `inclusion` validator.
- Allow the user to select their language using a select box with available/supported languages.
- All flash messages should be localized, like we did for the flash message on deletion of an article. If you can, make them having variable parts.
- Make sure that the articles index page displays the timestamps of the article creations using localization.
- Make sure that you application declares a default time zone inside the `config/application.rb`.
- Your `users` table needs to have the time zone table.
- Your `User` model needs to have validation on time zone so that the time zone takes values from specific set (using `inclusion` validator).
- Create the necessary UI on application layout to have the select box that would allow user to set their time zone.
- Create the around filter in order to take user time zone into account.

Important: Your code needs to be uploaded to your Github account. Also, your production application instance on Heroku should be updated accordingly.

10 - Background Jobs

Summary

Background Jobs - Summary

This is one of the most practical and useful chapters in the course. It introduces you to the idea of background jobs and background workers. The background asynchronous job execution is very common in Web development and you should be using it as much as possible, in order to minimize the HTTP request processing loop time. Definitely, a tool that you need to master.

Learning Goals

1. Learn how useful background jobs are in Web development.
2. Learn about the pattern to create a background job and shorten the HTTP request processing loop.
3. Learn about the most important properties of the background workers.
4. Learn about ActiveJob.
5. Learn how to define a background job.
6. Learn how to enqueue a background job.
7. Learn how to invoke the background worker.
8. Learn about Delayed Job.
9. Learn about different queues that you can use for your background jobs.
10. Learn how to start different background runners to consume different queues in parallel execution.

11. Learn about the background job life cycle callbacks.
12. Learn how Mailer actions can be executed in the background.
13. Learn how you can write automated tests for your background jobs.
14. Learn how you can schedule a background job to run at specific point of time in the future.
15. Learn about enabling the background runners in your Heroku production environment.

Introduction

Background jobs are very useful because they will allow us to run long-running tasks without consuming resources from our Web application that serves HTTP requests.

For example, when we want to send out an email, we will be creating a background job that would send the email out **asynchronously**. Hence, the HTTP request that generates the email is not blocked.

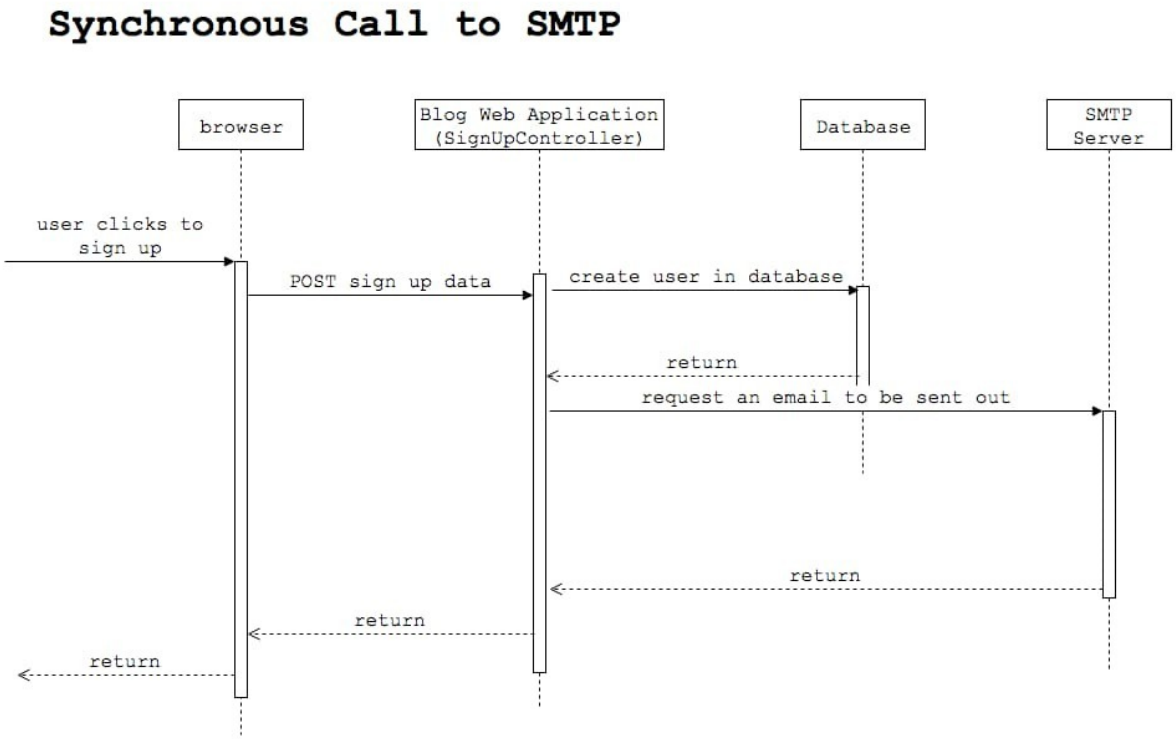
Think about the sign up process in our `blog` application. Do you remember the code that we have on successful sign up?

```
10
11  def create
12    user = User.new(user_params)
13    if user.save
14      send_email_to_confirm_their_email_address(user)
15      flash[:success] = 'You have successfully signed up! Please, confirm your email address. We just
16      redirect_to root_url
17    else
18      @user = user
19      render :new, status: :unprocessable_entity
20    end
21  end
22
```

this line of code will send an email synchronously!

Sign Up Synchronous Call To Send Email Out

The method `send_email_to_confirm_their_email_address` is calling method `UserMailer.ask_for_email_confirmation(...).deliver`, which makes a call to the SMTP/outgoing email server in order to send the email and return back the control of flow to the `SignUpController`.



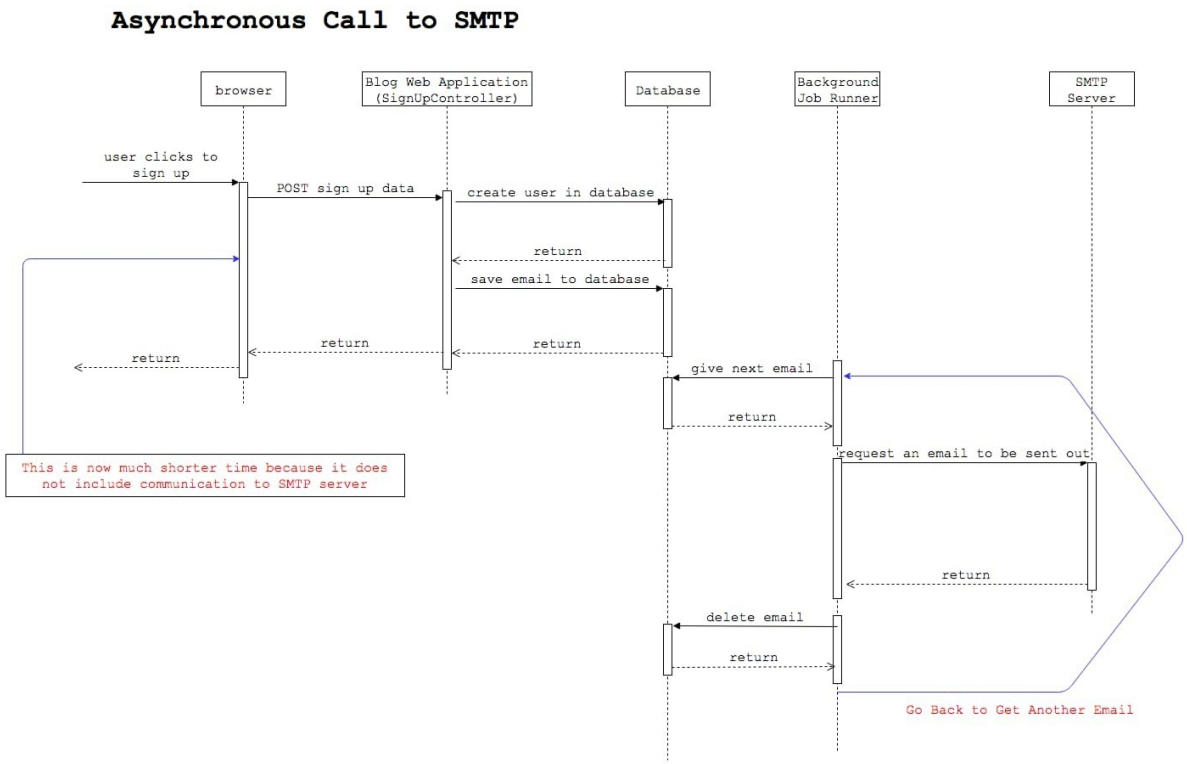
SMTP is being called synchronously within the HTTP request handling

You can see that the SMTP call to send out the email is within the loop of the HTTP request handling. The fact that the SMTP server is an external service and it takes too long both to contact and to actually do the work, means that the actual HTTP request will take long as well.

Also, if the SMTP server is down for any reason, the sign up will fail. However, we don't believe that failing to sign up due to inability to send the sign up confirmation email does worth it. We should find a way to send the email when the SMTP server will be up and running again and let the sign up process finish without error.

Introducing Background Runner

In order to solve the above problem, we are going to introduce another process that will be executing the delivery of the email outside of the main HTTP request processing loop.



Email Delivery Is the Responsibility of the Background Runner

Do you see the difference to the previous diagram. Now, the `SignUpController` saves the email (actually, it saves the task details to send an email out, which also includes the email itself) in the database, rather than talking to the SMTP server. And then returns immediately. This makes the HTTP request processing loop much shorter in terms of time. Also, there is no way the HTTP request fails due to SMTP server being down.

Then, it is the background runner that needs to pick up the details of the task from the database and execute what the task is supposed to be about. The email task will make the background runner send the email out by connecting to SMTP server. Then the email sent out will be deleted from the database and the background runner will loop back to pick up the next email in the queue.

Some Properties Of Background Runner

Background runners are Ruby processes that they have all the application code loaded, so they have access to the model and whatever else your Web application has access to. They run independently of your Web server, hence they may be down, while your server may be still running, and vice versa.

Usually, background runner is designed in such a way that it deletes only the successful tasks from the queue and keeps the failed ones inside the queue. Also, for the failed ones it is designed to pick them up again and retry their execution.

ActiveJob

`ActiveJob` is the Rails framework that offers an API to

1. create background jobs
2. enqueue background jobs in the backend database
3. execute background jobs

Also, it allows you to hook various background jobs technologies and use whatever matches your preferences.

Examples of background job technologies are

1. Delayed Job.
2. Resque.
3. Sidekiq.

These technologies are compatible with the `ActiveJob` interface and hence they can be plugged in interchangeably, without you having to change your application code.

> *Note* that Rails itself has an internal in-process asynchronous background job runner. It runs in its own thread, but keeps its queue with the background tasks in memory. This means that if the Rails server is restarted, any pending background task is lost.

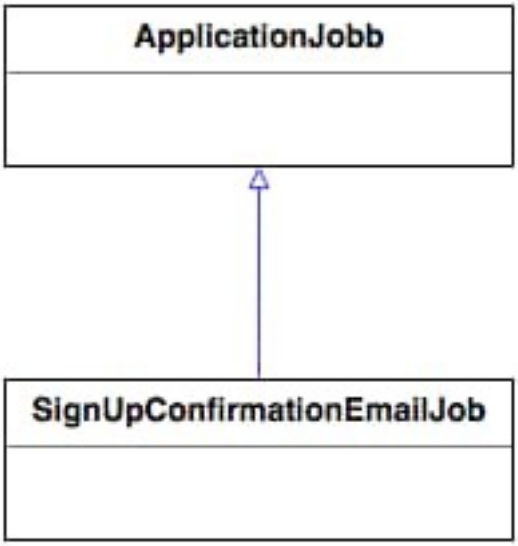
Define a Job

Let's define our first job. This is going to be a class that derives from `ApplicationJob`. Our first job is about sending the sign up confirmation email. We are going to call the class `SignUpConfirmationEmailJob` and we will define it inside the file `app/jobs/sign_up_-confirmation_email_job.rb`:

```
# File: app/jobs/sign_up_confirmation_email_job.rb
#
class SignUpConfirmationEmailJob < ApplicationJob
end
```

(the above code snippet online)

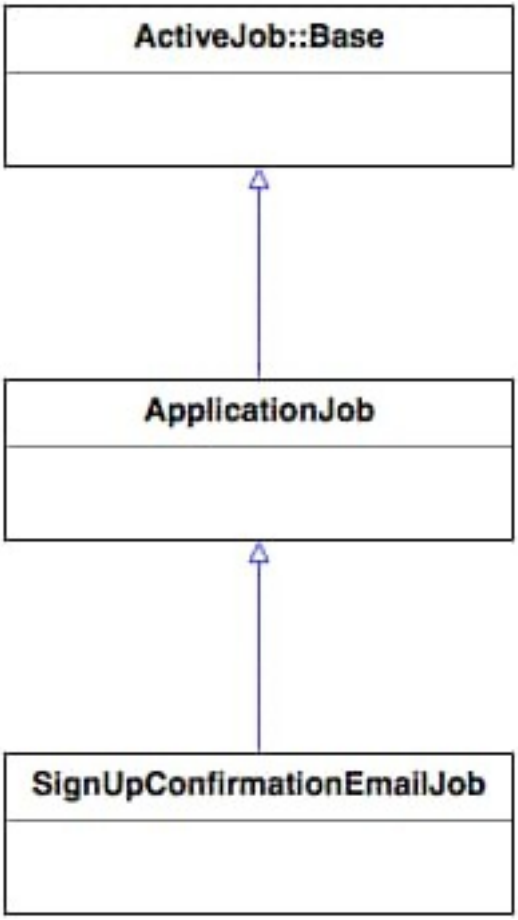
As you can see, we have the following class inheritance here:



Our Job Class Derives from ApplicationJob

The class `ApplicationJob` is actually a class inside our `app/jobs` folder that is generated there when we first set up our application. If you look at its code, it is an empty class. It just derives from `ActiveJob::Base`. The idea is that if you have things that are common to all your jobs, then we can put them in `ApplicationJob` class.

So, the actually hierarchy is this:



Top class is ActiveJob::Base

So, you can always visit the ActiveJob::Base API in order to study what this class offers to our Jobs.

`perform`

The jobs need to define a method with name `perform`. This method can take as many arguments as it needs. There is no restriction here. Here is the implementation of our `SignUpConfirmationEmailJob`:

```
# File: app/jobs/sign_up_confirmation_email_job.rb
#
class SignUpConfirmationEmailJob < ApplicationJob
  def perform(user_id)
    user = User.find(user_id)
    UserMailer.ask_for_email_confirmation(user.id).deliver
  end
end
```

(the above code snippet online)

Our job has a `perform` method that takes as input the id of the `User` that has just signed up. Hence, it will search for the user in the database and then ask the `UserMailer` to deliver the email.

All the method execution is going to take place in the background. We will explain how next.

Enqueue The Job

Now that we have the job definition, we can instantiate such jobs and put them in the background runner queue. This is called *enqueuing*.

We will change our `SignUpController#create` method to enqueue a job of type `SignUpConfirmationEmailJob`. The change is very simple:

```ruby
# File: app/controllers/sign_up_controller.rb
#
class SignUpController < ApplicationController
...
  def send_email_to_confirm_their_email_address(user)
    SignUpConfirmationEmailJob.perform_later(user.id)
  end
end
```

(the above code snippet online)

We changed the implementation of the method `send_email_to_confirm_their_email_address`. Instead of calling the `UserMailer.ask_for_email_confirmation(user.id).deliver`, we are now enqueuing a job of type `SignUpConfirmationEmailJob`.

That was it!

Now, we have implemented the asynchronous delivery of sign up confirmation emails. You only have to try that out.

Other Backends

Rails comes with an in-process and in-memory queue for asynchronous job execution. However, this is not for production. For production you will want to use a production-purpose backend. Also, you may want to install this backend at your development environment too and have a more detailed experience about how this backend works.

Rails has built-in adapters for many production-ready background job runners. See the full list here.

Delayed Job Backend

We will use delayed job for our backend. Let's switch it on:

Configure Application To Use Delayed Job

We need to configure our `blog` application to use Delayed Job. Here is how we do that:

```
# File: config/application.rb
#
...

    config.active_job.queue_adapter = :delayed_job
  end
end
```

(the above code snippet online)

Do you see line 25? This is how we say Rails that it should use the Delayed Job adapter.

Install Delayed Job

But still, enabling the Delayed Job adapter is not enough. You have to install Delayed Job. Since Delayed Job can use different backends for storing the job instances, but we want to use our database via the `ActiveRecord`, we have to install the gem delayed_job_active_record. Let's add this to our Gemfile and then run `bundle`. After you do that, you will see Delayed Job related gems to have been installed:

```
blog $ bundle show | grep 'delayed'
  * delayed_job (4.1.3)
  * delayed_job_active_record (4.1.2)
blog $
```

(the above code snippet online)

Generate The Schema Migration

Delayed Job with ActiveRecord backend is using our database to store the background jobs. It is using a specific table called `delayed_jobs`. This means that we have to create the corresponding schema migration and execute it.

The following command creates the schema migration file for Delayed Job:

```
blog $ rails generate delayed_job:active_record
      create  bin/delayed_job
       chmod  bin/delayed_job
      create  db/migrate/20170728185100_create_delayed_jobs.rb
blog $
```

(the above code snippet online)

Here is the schema migration file that is generated:

```
# File: db/migrate/20170728185100_create_delayed_jobs.rb
#
class CreateDelayedJobs < ActiveRecord::Migration[5.1]
  def self.up
    create_table :delayed_jobs, force: true do |table|
      table.integer :priority, default: 0, null: false # Allows some jobs to jump\
 to the front of the queue
      table.integer :attempts, default: 0, null: false # Provides for retries, bu\
t still fail eventually.
      table.text :handler,                 null: false # YAML-encoded string of t\
he object that will do work
      table.text :last_error                           # reason for last failure \
(See Note below)
      table.datetime :run_at                           # When to run. Could be Ti\
me.zone.now for immediately, or sometime in the future.
      table.datetime :locked_at                        # Set when a client is wor\
king on this object
      table.datetime :failed_at                        # Set when all retries hav\
e failed (actually, by default, the record is deleted instead)
      table.string :locked_by                          # Who is working on this o\
bject (if locked)
      table.string :queue                              # The name of the queue th\
is job is in
      table.timestamps null: true
    end

    add_index :delayed_jobs, [:priority, :run_at], name: "delayed_jobs_priority"
  end

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

(the above code snippet online)

And now let's run it:

```
blog $ bin/rake db:migrate
== 20170728185100 CreateDelayedJobs: migrating ================================
-- create_table(:delayed_jobs, {:force=>true})
   -> 0.0208s
-- add_index(:delayed_jobs, [:priority, :run_at], {:name=>"delayed_jobs_priority"\
})
   -> 0.0041s
== 20170728185100 CreateDelayedJobs: migrated (0.0251s) =======================

blog $
```

(the above code snippet online)

And let's confirm the creation of the table in the database:

```
blog $ psql -d blog_development -c "\d+ delayed_jobs"
                                                        Table "public.delayed_jo\
bs"
   Column   |            Type             |                        Modifiers    \
                     | Storage  | Stats target | Description
------------+-----------------------------+-------------------------------------\
---------------------+----------+--------------+-------------
 id         | bigint                      | not null default nextval('delayed_job\
s_id_seq'::regclass) | plain    |              |
 priority   | integer                     | not null default 0                   \
                     | plain    |              |
 attempts   | integer                     | not null default 0                   \
                     | plain    |              |
 handler    | text                        | not null                             \
                     | extended |              |
 last_error | text                        |                                      \
                     | extended |              |
 run_at     | timestamp without time zone |                                      \
                     | plain    |              |
 locked_at  | timestamp without time zone |                                      \
                     | plain    |              |
 failed_at  | timestamp without time zone |                                      \
                     | plain    |              |
 locked_by  | character varying           |                                      \
                     | extended |              |
 queue      | character varying           |                                      \
                     | extended |              |
 created_at | timestamp without time zone |                                      \
                     | plain    |              |
 updated_at | timestamp without time zone |                                      \
                     | plain    |              |
Indexes:
```

```
    "delayed_jobs_pkey" PRIMARY KEY, btree (id)
    "delayed_jobs_priority" btree (priority, run_at)
```

(the above code snippet online)

Try Delayed Job

With everything in place, let's try to sign up a user. Make sure that you restart your server.

When you carry out a sign up, this is what you will see in the Rails logs:

```
Started POST "/en/sign_up" for 127.0.0.1 at 2017-07-28 22:07:30 +0300
Processing by SignUpController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"...", "user"=>{"email"=>"paul1\
007@gmail.com", "password"=>"[FILTERED]", "password_confirmation"=>"[FILTERED]", \
"country_id"=>"6", "terms_of_service"=>"1"}, "commit"=>"Sign Up", "locale"=>"en"}
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS NULL LI\
MIT $1  [["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS N\
ULL LIMIT $1  [["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS N\
ULL LIMIT $1  [["LIMIT", 1]]
   (0.2ms)  BEGIN
  User Exists (0.5ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "paul1007@gmail.com"], ["LIMIT", 1]]
  SQL (0.8ms)  INSERT INTO "users" ("email", "password_digest", "created_at", "up\
dated_at", "email_confirmation_token", "terms_of_service", "country_id") VALUES (\
$1, $2, $3, $4, $5, $6, $7) RETURNING "id"  [["email", "paul1007@gmail.com"], ["p\
assword_digest", "$2a$10$Om9KkIY0SVwRiuEqy.KAMeCBtmWNb.4N3it71sjGcQOKK81RaCIn2"],\
 ["created_at", "2017-07-28 19:07:30.072978"], ["updated_at", "2017-07-28 19:07:3\
0.072978"], ["email_confirmation_token", "SHSBPxIIjqOW7mJSQsVO4g"], ["terms_of_se\
rvice", "t"], ["country_id", 6]]
   (1.0ms)  COMMIT
[ActiveJob]   (0.2ms)  BEGIN
[ActiveJob]  SQL (1.8ms)  INSERT INTO "delayed_jobs" ("handler", "run_at", "queu\
e", "created_at", "updated_at") VALUES ($1, $2, $3, $4, $5) RETURNING "id"  [["ha\
ndler", "--- !ruby/object:ActiveJob::QueueAdapters::DelayedJobAdapter::JobWrapper\
\njob_data:\n  job_class: SignUpConfirmationEmailJob\n  job_id: a14bf4fd-8169-406\
9-9f86-cb8b3a7d2700\n  queue_name: default\n  priority: \n  arguments:\n  - 33\n \
 executions: 0\n  locale: en\n"], ["run_at", "2017-07-28 19:07:30.122225"], ["que\
ue", "default"], ["created_at", "2017-07-28 19:07:30.122299"], ["updated_at", "20\
17-07-28 19:07:30.122299"]]
[ActiveJob]   (0.3ms)  COMMIT
[ActiveJob] Enqueued SignUpConfirmationEmailJob (Job ID: a14bf4fd-8169-4069-9f86-\
cb8b3a7d2700) to DelayedJob(default) with arguments: 33
Redirected to http://localhost:3000/en
Completed 302 Found in 124ms (ActiveRecord: 10.4ms)
```

(the above code snippet online)

Do you see line 12? It is an `insert into "delayed_jobs"`. This is when Rails stores the background job into the queue of Delayed Job. Also, see log line 14. It is mentioning that the job of type `SignUpConfirmationEmailJob` has been enqueued inside `DelayedJob(default)`, i.e. inside the `DelayedJob` backend.

The `default` here meaning the `default` queue.

But, you don't see any log related to email delivery. Also, you should not see any email inside your SMTP Mailtrap email box. This is because the background job runner has not picked up the job from the queue. And it has not picked it up because the background runner is not running.

You can also confirm that there is one item in the queue as follows:

```
1  blog $ psql -d blog_development -c "select * from delayed_jobs"
2   id | priority | attempts |                                handler              \
3                     | last_error |           run_at           | locked_at | faile\
4  d_at | locked_by |  queue   |         created_at         |         updated_at     \
5
6  ----+----------+----------+-------------------------------------------------------\
7  --------------------+------------+----------------------------+-----------+------\
8  -----+-----------+---------+----------------------------+------------------------\
9  ----
10   1 |        0 |        0 | --- !ruby/object:ActiveJob::QueueAdapters::DelayedJob\
11 Adapter::JobWrapper+|            | 2017-07-28 19:07:30.122225 |           |      \
12      |           | default | 2017-07-28 19:07:30.122299 | 2017-07-28 19:07:30.122\
13 299
14     |          |          | job_data:                                            \
15                    +|            |                            |           |      \
16      |           |         |                            |
17     |          |          |   job_class: SignUpConfirmationEmailJob              \
18                    +|            |                            |           |      \
19      |           |         |                            |
20     |          |          |   job_id: a14bf4fd-8169-4069-9f86-cb8b3a7d2700       \
21                    +|            |                            |           |      \
22      |           |         |                            |
23     |          |          |   queue_name: default                                \
24                    +|            |                            |           |      \
25      |           |         |                            |
26     |          |          |   priority:                                          \
27                    +|            |                            |           |      \
28      |           |         |                            |
29     |          |          |   arguments:                                         \
30                    +|            |                            |           |      \
31      |           |         |                            |
32     |          |          |   - 33                                               \
33                    +|            |                            |           |      \
34      |           |         |                            |
```

```
    |          |          |   executions: 0                                  \
                    +|            |                           |           |      \
     |           |         |                           |
    |          |          |   locale: en                                     \
                    +|            |                           |           |      \
     |           |         |                           |
    |          |          |                                                   \
                     |            |                           |           |      \
     |           |         |                           |
(1 row)

blog $
```

(the above code snippet online)

So, we need a background runner to start and consume this queue item.

How to Start the Background Runner

The Delayed Job has installed an executable inside `bin`, that allows you to start and stop the background runner. Let's start it:

```
blog $ bin/delayed_job start
...rvm/gems/ruby-2.3.3@blog/gems/delayed_job-4.1.3/lib/delayed/command.rb:5:in `r\
escue in <top (required)>': You need to add gem 'daemons' to your Gemfile if you \
wish to use it. (RuntimeError)
        from ...rvm/gems/ruby-2.3.3@blog/gems/delayed_job-4.1.3/lib/delayed/comma\
nd.rb:2:in `<top (required)>'
        from ...rvm/gems/ruby-2.3.3@blog/gems/activesupport-5.1.1/lib/active_supp\
ort/dependencies.rb:292:in `require'
        from ...rvm/gems/ruby-2.3.3@blog/gems/activesupport-5.1.1/lib/active_supp\
ort/dependencies.rb:292:in `block in require'
        from ...rvm/gems/ruby-2.3.3@blog/gems/activesupport-5.1.1/lib/active_supp\
ort/dependencies.rb:258:in `load_dependency'
        from ...rvm/gems/ruby-2.3.3@blog/gems/activesupport-5.1.1/lib/active_supp\
ort/dependencies.rb:292:in `require'
        from bin/delayed_job:4:in `<main>'
blog $
```

(the above code snippet online)

As you can see, in order to be able to use the Delayed Job background runner, we need the `daemons` gem installed. Let's add it to the Gemfile, and execute a `bundle`.

Then try `bin/delayed_job start` again:

```
blog $ bin/delayed_job start
delayed_job: process with pid 36553 started.
blog $
```

(the above code snippet online)

You can see that a new process with id 36553 has started. This is the background runner. As soon as the background runner starts, it starts consuming queue items, i.e. jobs from the database. Hence, if you now check the background jobs in the database, you will see that there is none:

```
blog $ psql -d blog_development -c "select * from delayed_jobs"
 id | priority | attempts | handler | last_error | run_at | locked_at | failed_at\
 | locked_by | queue | created_at | updated_at
----+----------+----------+---------+------------+--------+-----------+----------\
-+-----------+-------+------------+------------
(0 rows)

blog $
```

(the above code snippet online)

The job has been executed and the sign up email has been sent out.

If you want to stop the background runner from running, you can issue the command `bin/delayed_job stop`.

Configure Delayed Job

It is a good practice to configure Delayed Job using an initializer file like this:

```
# File: config/initializers/delayed_job_initializer.rb
#
Delayed::Worker.destroy_failed_jobs = false
Delayed::Worker.sleep_delay = 5
Delayed::Worker.max_attempts = 3
Delayed::Worker.max_run_time = 5.minutes
Delayed::Worker.read_ahead = 10
Delayed::Worker.default_queue_name = 'default'
Delayed::Worker.delay_jobs = !Rails.env.test?
Delayed::Worker.raise_signal_exceptions = :term
Delayed::Worker.logger = Logger.new(File.join(Rails.root, 'log', "delayed_job_#{R\
ails.env}.log"))
```

(the above code snippet online)

Save the above file and restart your Rails server and your background runners. Please, note that you may want to have different settings among different environments. For example, the `max_run_time` might different in production from development environment.

Queues

All the background jobs are being enqueued inside the job runner queue. For Delayed Job, this is the `delayed_jobs` table. However, This queue may further be subdivided to other logical queues. This is the reasons you see the column `queue` as part of the columns of the `delayed_jobs` table.

You can define the queue at the job definition class level:

```
1  # File: app/jobs/sign_up_confirmation_email_job.rb
2  #
3  class SignUpConfirmationEmailJob < ApplicationJob
4    queue_as :low_priority
5
6    def perform(user_id)
7      user = User.find(user_id)
8      UserMailer.ask_for_email_confirmation(user.id).deliver
9    end
10 end
```

(the above code snippet online)

Do you see the line 4? This is how we tell Rails that each instance of this job will be queued inside the queue with name `low_priority`. Let's try to create a new sign up job by signing up a new user. Please, make sure that the delayed job background runner is stopped (`bin/delayed_job stop`).

As soon as you sign up a new user, you can check what it is inside the queue. For example:

```
1  blog $ psql -d blog_development -c "select * from delayed_jobs"
2   id | priority | attempts |                                handler              \
3                     | last_error |          run_at           | locked_at | failed\
4  _at | locked_by |    queue     |         created_at         |         updated_at \
5
6  ----+----------+----------+-------------------------------------------------------\
7  --------------------+------------+--------------------------+-----------+-------\
8  ----+-----------+--------------+--------------------------+--------------------\
9  --------
10   4 |        0 |        0 | --- !ruby/object:ActiveJob::QueueAdapters::DelayedJob\
11 Adapter::JobWrapper+|            | 2017-07-29 04:49:14.72993 |           |       \
12     |           | low_priority | 2017-07-29 04:49:14.729983 | 2017-07-29 04:49:14\
13 .729983
14     |          |          | job_data:                                          \
15                    +|            |                          |           |       \
16     |           |              |                          |
17     |          |          |   job_class: SignUpConfirmationEmailJob            \
18                    +|            |                          |           |       \
19     |           |              |                          |
20     |          |          |   job_id: f136d1e1-214b-4a2f-936c-e27b00360d9a      \
```

```
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |   queue_name: low_priority                          \
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |   priority:                                         \
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |   arguments:                                        \
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |   - 36                                              \
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |   executions: 0                                     \
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |   locale: en                                        \
                    +|            |                          |           |       \
    |           |              |                          |
    |          |          |                                                     \
                     |            |                          |           |       \
    |           |              |                          |
(1 row)

blog $
```

(the above code snippet online)

Do you see the column with name `queue`? It has the value `low_priority`. Now, let's start the background runner, but now, with specifying which queue the background runner should be dealing with. Let's give the name `default`.

```
blog $ bin/delayed_job --queue default start
delayed_job: process with pid 39314 started.
blog $
```

(the above code snippet online)

Even if you wait quite long, the queue will not be processed by this particular background runner. Just try, after a while, to check the contents of the `delayed_jobs` table. They will still be there. This is because the background runner that we have started does not deal with the queue of the item that exists in the `delayed_jobs` table.

We can now start a background runner for the `low_priority` queue:

```
blog $ bin/delayed_job --queue low_priority start
ERROR: there is already one or more instance(s) of the program running
blog $
```

(the above code snippet online)

The above command failed because Delayed Job is already running for the `default` queue. But, you can still start another worker running in parallel to the previous one. You just need to give a different identifier to the new worker:

```
blog $ bin/delayed_job --queue low_priority --identifier=low_priority_1 start
delayed_job.low_priority_1: process with pid 39704 started.
blog $
```

(the above code snippet online)

With the `--identifier` option we uniquely identify the new process to spawn. And the spawn was successful. Now, if you check the number of entries in the `delayed_jobs` table, you will see them being zero.

```
blog $ psql -d blog_development -c "select * from delayed_jobs"
id | priority | attempts | handler | last_error | run_at | locked_at | failed_at \
| locked_by | queue | created_at | updated_at
----+----------+----------+---------+------------+--------+-----------+----------\
-+-----------+-------+------------+------------
(0 rows)

blog $
```

(the above code snippet online)

Also, if you now do list the processes that are related to Delayed Job, you will find two of them running:

```
blog $ ps -ef | grep 'delayed_job'
  501 39452     1   0  7:49AM ??         0:01.58 delayed_job
  501 39704     1   0  7:56AM ??         0:01.25 delayed_job.low_priority_1
...
blog $
```

(the above code snippet online)

If you want to stop these workers from running...

```
blog $ bin/delayed_job stop
delayed_job: trying to stop process with pid 39452...
delayed_job: process with pid 39452 successfully stopped.
blog $ bin/delayed_job --identifier=low_priority_1 stop
delayed_job.low_priority_1: trying to stop process with pid 39704...
delayed_job.low_priority_1: process with pid 39704 successfully stopped.
blog $
```

(the above code snippet online)

> *Information:* You can always find more about how to start Delayed Job workers in the Delayed Job gem documentation pages. Or you can issue `bin/delayed_job --help` and get the list of options that the executable can take.

Callbacks

ActiveJob allows you to hook code at specific points in the lifecycle of a job. These are called ActiveJob Callbacks and they are the following:

1. Enqueue Callbacks
    1. before_enqueue
    2. around_enqueue
    3. after_enqueue
2. Perform Callbacks

    1. before_perform
    2. around_perform
    3. after_perform

Let's implement all of them for our `SignUpConfirmationEmailJob`:

```ruby
# File: app/jobs/sign_up_confirmation_email_job.rb
#
class SignUpConfirmationEmailJob < ApplicationJob
  queue_as :low_priority

  before_enqueue do |job|
    puts "Before enqueuing job: #{job.inspect}"
  end

  around_enqueue do |job, enqueue_block|
    puts "Before executing enqueue in an around callback, #{job.inspect}"

    enqueue_block.call
```

```
14
15      puts "After executing enqueue in an around callback, #{job.inspect}"
16    end
17
18    after_enqueue do |job|
19      puts "After enqueuing job: #{job.inspect}"
20    end
21
22    before_perform do |job|
23      puts "Before performing job: #{job.inspect}"
24    end
25
26    around_perform do |job, perform_block|
27      puts "Before executing perform in an around callback, #{job.inspect}"
28
29      perform_block.call
30
31      puts "After executing perform in an around callback, #{job.inspect}"
32    end
33
34    after_perform do |job|
35      puts "After performing job: #{job.inspect}"
36    end
37
38    def perform(user_id)
39      user = User.find(user_id)
40      UserMailer.ask_for_email_confirmation(user.id).deliver
41    end
42  end
```

(the above code snippet online)

Do you see how the callbacks are registered in an ActiveJob definition? Note that the non-around callbacks, the yield the job instance in the block. Whereas the around callbacks they yield both the job instance and the block to call (enqueue or perform). The around callbacks need to call the block, otherwise, the enqueue or the perform will not be executed.

With the above code in place, try to sign up a new user and watch your Rails console output:

```
Started POST "/en/sign_up" for 127.0.0.1 at 2017-07-29 08:30:00 +0300
Processing by SignUpController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"...", "user"=>{"email"=>"ian08\
29@gmail.com", "password"=>"[FILTERED]", "password_confirmation"=>"[FILTERED]", "\
country_id"=>"3", "terms_of_service"=>"1"}, "commit"=>"Sign Up", "locale"=>"en"}
  User Load (0.5ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS NULL LI\
MIT $1  [["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS N\
ULL LIMIT $1  [["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id" IS N\
ULL LIMIT $1  [["LIMIT", 1]]
   (0.2ms)  BEGIN
  User Exists (0.6ms)  SELECT  1 AS one FROM "users" WHERE LOWER("users"."email")\
 = LOWER($1) LIMIT $2  [["email", "ian0829@gmail.com"], ["LIMIT", 1]]
  SQL (1.0ms)  INSERT INTO "users" ("email", "password_digest", "created_at", "up\
dated_at", "email_confirmation_token", "terms_of_service", "country_id") VALUES (\
$1, $2, $3, $4, $5, $6, $7) RETURNING "id"  [["email", "ian0829@gmail.com"], ["pa\
ssword_digest", "$2a$10$AO8zaiXu3usptQRbxu0KJe3wkdOIm1xWTY4HmaHCQ4xJjhcG3n48i"], \
["created_at", "2017-07-29 05:30:00.595836"], ["updated_at", "2017-07-29 05:30:00\
.595836"], ["email_confirmation_token", "d-Xo4zcXojPzYRGGne58_A"], ["terms_of_ser\
vice", "t"], ["country_id", 3]]
   (0.3ms)  COMMIT
Before enqueuing job: #<SignUpConfirmationEmailJob:0x007fc9ccb2a348 @arguments=[3\
7], @job_id="cbf8801b-270c-4e05-964c-67e005fcfc41", @queue_name="low_priority", @\
priority=nil, @executions=0>
Before executing enqueue in an around callback, #<SignUpConfirmationEmailJob:0x00\
7fc9ccb2a348 @arguments=[37], @job_id="cbf8801b-270c-4e05-964c-67e005fcfc41", @qu\
eue_name="low_priority", @priority=nil, @executions=0>
[ActiveJob]    (0.2ms)  BEGIN
[ActiveJob]   SQL (0.6ms)  INSERT INTO "delayed_jobs" ("handler", "run_at", "queu\
e", "created_at", "updated_at") VALUES ($1, $2, $3, $4, $5) RETURNING "id"  [["ha\
ndler", "--- !ruby/object:ActiveJob::QueueAdapters::DelayedJobAdapter::JobWrapper\
\njob_data:\n  job_class: SignUpConfirmationEmailJob\n  job_id: cbf8801b-270c-4e0\
5-964c-67e005fcfc41\n  queue_name: low_priority\n  priority: \n  arguments:\n  - \
37\n  executions: 0\n  locale: en\n"], ["run_at", "2017-07-29 05:30:00.658290"], \
["queue", "low_priority"], ["created_at", "2017-07-29 05:30:00.658374"], ["update\
d_at", "2017-07-29 05:30:00.658374"]]
[ActiveJob]    (0.3ms)  COMMIT
After enqueuing job: #<SignUpConfirmationEmailJob:0x007fc9ccb2a348 @arguments=[37\
], @job_id="cbf8801b-270c-4e05-964c-67e005fcfc41", @queue_name="low_priority", @p\
riority=nil, @executions=0, @provider_job_id=5>
After executing enqueue in an around callback, #<SignUpConfirmationEmailJob:0x007\
fc9ccb2a348 @arguments=[37], @job_id="cbf8801b-270c-4e05-964c-67e005fcfc41", @que\
ue_name="low_priority", @priority=nil, @executions=0, @provider_job_id=5>
[ActiveJob] Enqueued SignUpConfirmationEmailJob (Job ID: cbf8801b-270c-4e05-964c-\
67e005fcfc41) to DelayedJob(low_priority) with arguments: 37
```

```
Redirected to http://localhost:3000/en
Completed 302 Found in 148ms (ActiveRecord: 8.8ms)
```

(the above code snippet online)

Do you see lines 11, 12, 16 and 17? These are the `puts` commands for the enqueue callbacks. You don't see any output related to the perform callbacks because the perform has not been carried out. Nevertheless, you will not see them in the Rails logs, but you may see them in the background runner logs when the background runner picks up the job to perform it.

Start the background runner:

```
blog $ bin/delayed_job start
delayed_job: process with pid 41304 started.
blog $
```

(the above code snippet online)

This will definitely run your job, but you will not see the `puts` commands of the perform hooks, because the background runner runs without terminal output.

What you can use instead of `puts` commands is the `Rails.logger` command. This is a normal logger that you can use to log message inside the file that Rails is using for logging too:

```ruby
# File: app/jobs/sign_up_confirmation_email_job.rb
#
class SignUpConfirmationEmailJob < ApplicationJob
  queue_as :low_priority

  before_enqueue do |job|
    puts "Before enqueuing job: #{job.inspect}"
  end

  around_enqueue do |job, enqueue_block|
    puts "Before executing enqueue in an around callback, #{job.inspect}"

    enqueue_block.call

    puts "After executing enqueue in an around callback, #{job.inspect}"
  end

  after_enqueue do |job|
    puts "After enqueuing job: #{job.inspect}"
  end

  before_perform do |job|
    Rails.logger.debug "Before performing job: #{job.inspect}"
  end
```

```

  around_perform do |job, perform_block|
    Rails.logger.debug "Before executing perform in an around callback, #{job.ins\
pect}"

    perform_block.call

    Rails.logger.debug "After executing perform in an around callback, #{job.insp\
ect}"
  end

  after_perform do |job|
    Rails.logger.debug "After performing job: #{job.inspect}"
  end

  def perform(user_id)
    user = User.find(user_id)
    UserMailer.ask_for_email_confirmation(user.id).deliver
  end
end
```

(the above code snippet online)

Do you see the perform callbacks? I am using `Rails.logger.debug` instead of `puts`. Restart your background runner and try to sign up a user again. Inside the `log/development.log` file, you will see lines like this:

```
[ActiveJob] [SignUpConfirmationEmailJob] [1c2e564a-0eeb-4f19-8869-6fecbbcbecdb] B\
efore performing job: #<SignUpConfirmationEmailJob:0x007ff71c594b40 @arguments=[3\
8], @job_id="1c2e564a-0eeb-4f19-8869-6fecbbcbecdb", @queue_name="low_priority", @\
priority=nil, @executions=0, @provider_job_id=nil, @serialized_arguments=nil, @lo\
cale="en">
```

(the above code snippet online)

Callbacks Have Access To Job Arguments

Before closing the subject of callbacks, see how the hooks have access to the job instance. The `job` variable also has the `@arguments` instance variable that is an array of the arguments passed to the perform method. That can be useful in many ways inside your hooks.

Mailers

Although we used the email example to demonstrate the definition of custom jobs, especially for mailers, you don't necessarily have to define an `ActiveJob` class. Rails provides the method `deliver_later` that allow you to enqueue a mailer job without any more effort.

Hence, the call:

```
UserMailer.ask_for_email_confirmation(user.id).deliver_later
```

(the above code snippet online)

can enqueue a `UserMailer` job in the background jobs without you having to have defined a custom job class for it.

Job Testing

Testing jobs is done with ruby code that we write inside `test/jobs` folder. Let's write a test for our `SignUpConfirmationEmailJob`:

```
# File test/jobs/sign_up_confirmation_email_job_test.rb
#
class SignUpConfirmationEmailJobTest < ActiveJob::TestCase
  test 'an email is put in the queue' do
    user = create(:user)

    ActionMailer::Base.deliveries.clear
    assert_difference "ActionMailer::Base.deliveries.size", 1 do
      SignUpConfirmationEmailJob.perform_now(user.id)
    end
  end
end
```

(the above code snippet online)

Here we are testing the perform of the job, by actually calling `.perform_now`. The `.perform_now` method can be called whenever you want the job to be executed immediately/synchronously and not in the background. A test is a good place for calling this method.

If you want to read more about Job Testing you can read the Rails Guides here.

Scheduling Jobs To Run At Specific Points in The Future

We have seen how you can put a job in the queue. Then, if a background runner is available to execute it, it picks it up immediately. However, there are cases in which you want your job to run at specific point of time in the future. For example:

```
MyBillingJob.set(wait_until: Time.zone.now + 2.days).perform_later(customer)
```

(the above code snippet online)

The above will put the job in the queue, but, even if there is a background runner running, it will not be executed until 2 days from now.

Background Runners in Heroku

Running background runners in Heroku means that you need to define a worker in your `Procfile`. The `Procfile` (you can read more about it here) is a file that needs to exist in the root folder of your project. This is what you have to define for your `blog`:

```
web: bundle exec rails server
worker: bundle exec rake jobs:work
```

(the above code snippet online)

With the above `Procfile` in place, when you deploy to Heroku, you will also enable a background runner that would be executing your jobs asynchronously.

> **Important**: Having a background runner in Heroku might increase the cost of your application. Make sure that you understand how Heroku charges you for the resources that you consume. You may want to disable Delayed Job for your production environment too, until you have your application finished and ready to serve real production traffic. Disabling Delayed Job in production environment is as simple as changing this `Delayed::Worker.delay_jobs = !Rails.env.test?` to `Delayed::Worker.delay_jobs = !Rails.env.test? && !Rails.env.production?`.

Closing Note

Background jobs and technologies to deploy this feature are very common in all Web applications. If you want to read more about ActiveJob you can read the Rails Guides here. You should definitely study Delayed Job as well as other tools like Sidekiq or Resque. And don't forget to read the Best Practices about Delayed Job here.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz Details

The quiz for this chapter can be found here

Task Details

You need to make sure that you have implemented in your `blog` application the things that the chapter teaches you. Indicatively:

1. Create the `SignUpConfirmationEmailJob` and make the email being delivered asynchronously.
2. Install and use the Delayed Job with the ActiveRecord adapter.
3. Set the queue to be `low_priority`.
4. Write sample code for all the callbacks.
5. Write at least one test for your job.

Important: Upload your work to your Github account. Also, when you deploy your work to Heroku make sure that you understand whether you are charged or not for your background workers.

11 - JavaScript with Rails Applications

Summary

JavaScript With Rails Application - Summary

You already know JavaScript and jQuery. This chapter is going to teach you, firstly, about, Ajax, which is one of the most important technologies in the JavaScript world. Ajax is used in every JavaScript framework and it is not dependent on Ruby on Rails. However, RoR has JavaScript and Ajax integration that can make your life much easier and more productive. You will learn about all these features. On top of that, you will also learn about WebSockets and how they have revolutionized the way browsers communicate with servers, including Ruby on Rails. RoR has also developed a Ruby layer around WebSockets that abstracts the underlying technology and makes very easy for you to implement full-duplex browser to server communication.

Learning Goals

1. Learn about Ajax.
2. Learn how to bind an Ajax request to a link or a button.
3. Learn how to handle the success case of an Ajax request.
4. Learn how to handle the error case of an Ajax request.
5. Learn about the jQuery support of Ajax.
6. Learn how to update parts of your page based on the response you receive from an Ajax request.
7. Learn how to adapt your controller to deal with Ajax requests.
8. Learn what Rails offers to us with regards to Ajax requests.
9. Learn about WebSockets.

10. Learn how Rails integrates WebSockets technology into your Web app using ActionCable.
11. Learn about ActionCable Connections.
12. Learn about ActionCable Channels.
13. Learn about ActionCable Consumers and Subscribers.
14. Learn about the Publish / Subscriber communication technique.
15. Learn how to hook code at the moment a new connection is accepted.
16. Learn about the class all Channels derive from.
17. Learn how to hook code at the moment a new subscription takes place.
18. Learn how to write JavaScript that would be creating the Consumer and registering a Subscription.
19. Learn how to inspect the Rails and Developer Tools Network tab logs.
20. Learn about the broadcasting steams and how they are used to broadcast messages to all subscribers.
21. Learn about the subscription adapters.
22. Learn about Redis and how you can use it as your subscription adapter in your development environment.
23. Learn how to set your Subscribers ready to handle the published messages.
24. Learn how a Consumer can publish a message for all the Subscribers to receive.
25. Learn how to uniquely identify each Subscriber connection.
26. Learn how to push a message coming in from a Subscriber to all other Subscribers, excluding the sender itself.

Introduction

We have learned to program in JavaScript and we have also learned how to use jQuery library. This gave us the ability to change how the front-end, the browser-end, of our application behaves.

But JavaScript is used to communicate with the server too, Ruby on Rails in our case. The technique used is called *Ajax* (also *AJAX*) which stands for Asynchronous JavaScript and XML.

Introduction to Ajax

The basic need for Ajax is when we want to send a request from the browser to the server in order to get some pieces of information back as a response, but not the whole page content. The request is usually done asynchronously, i.e. without blocking the user interaction to the page. The pieces of information returned are used to update parts of the page. This makes it a very efficient way of exchanging information with the server, because very few information is travelling from server back to the browser.

> *Information:* Rails JavaScript integration is supporting CoffeeScript too. Your coffee script is a script that finally compiles to JavaScript. We will not be using CoffeeScript in this course since Rails allows us to write JavaScript too. It does not oblige us to write CoffeeScript. We encourage you to start learning CoffeeScript though, because it is becoming very popular.

jQuery and Ajax

Hence, Ajax is used to send HTTP requests to server. jQuery comes with Ajax support and makes it really easy for you to build Ajax requests.

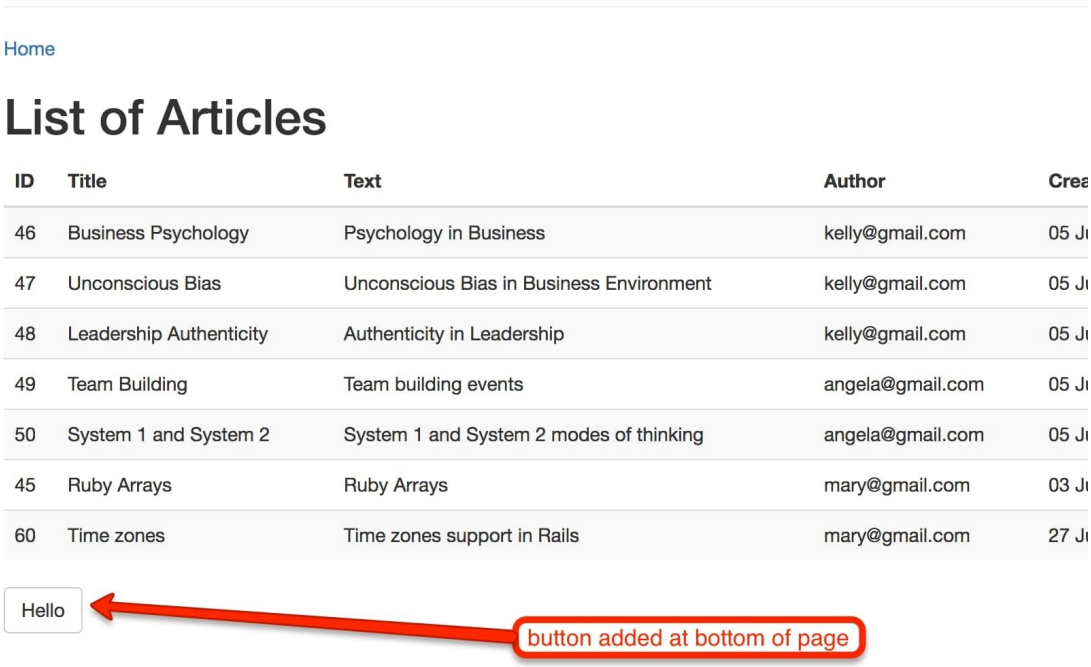
Button To Trigger Request

Let's see an example. Let's add a `Hello` button at the bottom of our application layout in our `blog` application:

```
- # File: app/views/layouts/application.html.haml
- #
!!!
%html
...
      = yield :menu
      = yield

      = button_tag 'Hello', id: 'hello-button', class: 'btn btn-default'
```

(the above code snippet online)

If you save and load a page, e.g. http://localhost:3000/articles, it will have the button at the bottom of your page:



Hello Button At the Bottom Of The Page

If you try to click this button, nothing will happen. This is because this is not a submit button as part of a form.

But still, we can use jQuery and Ajax to submit a GET request to the server. We will send an HTTP request to the endpoint `GET /hello`.

Trigger Request

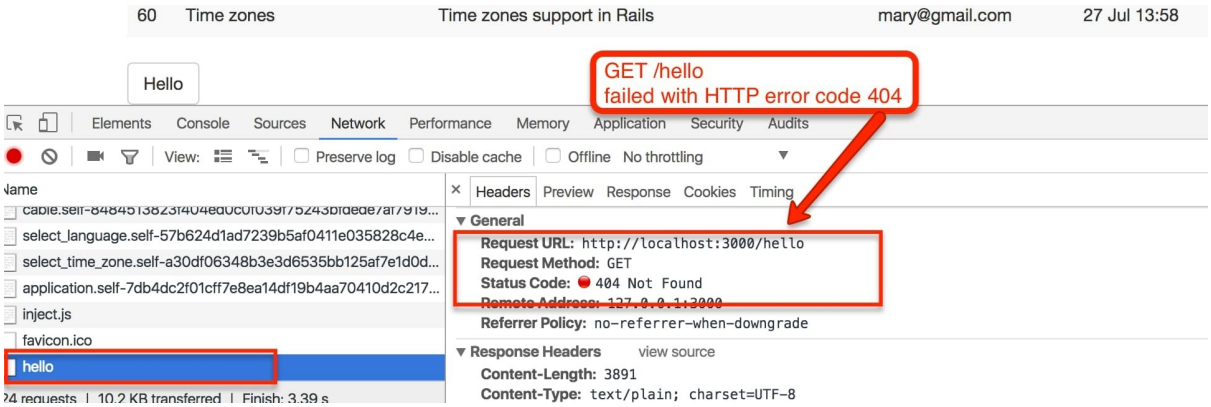
Open our `app/assets/javascripts/application.js` file and amend it so that it includes the following JavaScript code:

```
1  // File: app/assets/javascripts/application.js
2  //
3  // This is a manifest file that'll be compiled into application.js, which will in\
4  clude all the files
5  ...
6  //= require_tree .
7
8  $(document).ready(function() {
9      $('#hello-button').on('click', function() {
10         $.get({
11             url: '/hello',
12             success: function(data, textStatus, jqXHR) {
13                 console.log('textStatus: ' + textStatus);
14                 alert('Hello ' + data);
15             }
16         });
17     });
18 });
```

(the above code snippet online)

The new lines that we have added are lines 20 to 30. We are installing a `click` handler at the button with id `hello-button`. The handler is calling `$.get()` method which is the jQuery Ajax get method. This method takes as argument a JavaScript object. One property of the object is `url` and has the value of the path we want the GET request to be sent to. The other property is `success`. Its value is a function that will be called when the response from server comes back to the browser. So, it is called on HTTP success response. When `success` is called, then the first argument, `data` contains whatever the HTTP response body contains. The `textStatus` is going to contain the HTTP Response status.

If you save the above, reload the page on your browser and click on the `Hello` button, you will not see anything. Actually, if you have the `Network` tab open at your Google Chrome Developer Tools, you will see the GET request to `/hello` failing with error code 404.



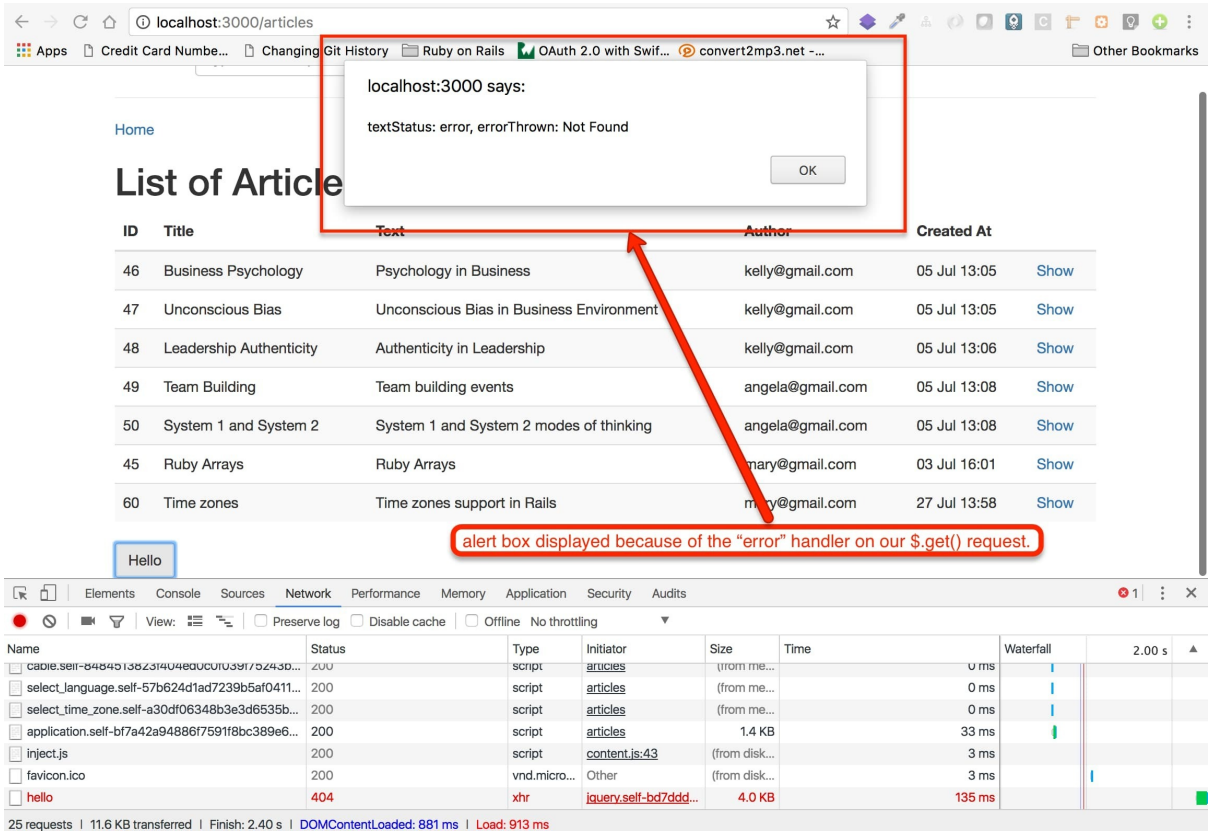
GET Failed with 404

Before we fix this, let's also attach a handler for error cases. This is the new `app/assets/javascripts/applicatio` file:

```
// File: app/assets/javascripts/application.js
...
//= require_tree .

$(document).ready(function() {
    $('#hello-button').on('click', function() {
        $.get({
            url: '/hello',
            success: function(data, textStatus, jqXHR) {
                console.log(textStatus);
                alert('Hello ' + data);
            },
            error: function(jqXHR, textStatus, errorThrown) {
                alert('textStatus: ' + textStatus + ', errorThrown: ' + errorThro\
wn);
            }
        });
    });
});
```

(the above code snippet online)

After the `success` handler, we have defined the `error` handler too. Save the above, reload your page on the browser and try to click the button `Hello`. You will see this:



Error Handler Displays Alert

Obviously, that was expected. We have not implemented the end point `GET /hello` and that is why the request fails. You can confirm the missing endpoint at the server side too, by inspecting the Rails logs:

```
Started GET "/hello" for 127.0.0.1 at 2017-07-31 08:25:25 +0300

ActionController::RoutingError (No route matches [GET] "/hello"):
```

(the above code snippet online)

Implementing The End Point

Let's implement the end point. Add the following entry in your `config/routes.rb` file:

```
1  # File: config/routes.rb
2  #
3  Rails.application.routes.draw do
4    post '/users/locale', to: 'users#update_locale'
5    post '/users/time_zone', to: 'users#update_time_zone'
6
7    scope "(:locale)", locale: Regexp.new(I18n.available_locales.map(&:to_s).join('\
8  |')) do
9      get '/hello', to: 'hello#respond'
10 ...
11   end
12 end
```

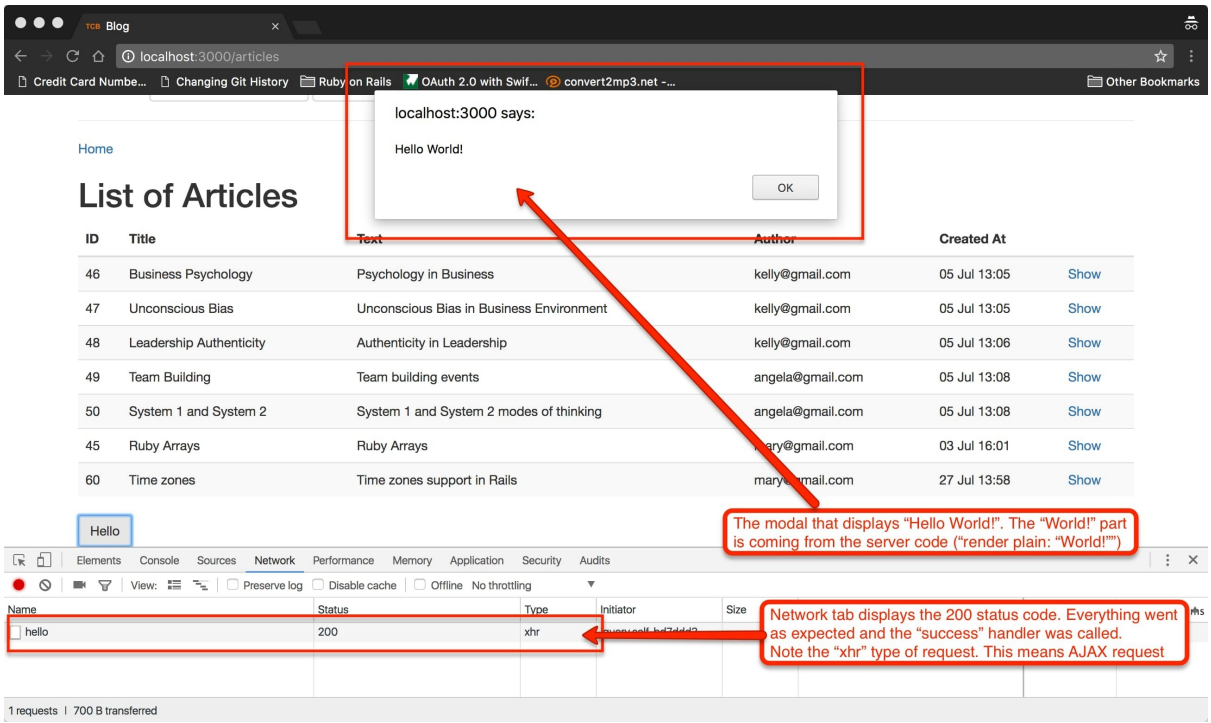
(the above code snippet online)

Do you see line 8? This is the new line that we have added. It defines that the `GET /hello` is going to be served by `HelloController#respond` action. Let's implement this too:

```
1  # File: app/controllers/hello_controller.rb
2  #
3  class HelloController < ApplicationController
4    skip_before_action :authenticate_user
5
6    def respond
7      render plain: 'World!'
8    end
9  end
```

(the above code snippet online)

The `respond` action is using the `render plain` to send back to the response a simple `World!` string. With this in place, let's click the button again:



Successful Ajax Request

`$.ajax` API

The `$.get` API is very useful if you want to issue a GET request. However, it relies on calling `$.ajax` internally. The `$.ajax` is the method that can be used to call any type of Ajax requests.

Let's convert the `$.get` to its `$.ajax` equivalent:

```
// File: app/assets/javascripts/application.js
...
//= require_tree .

$(document).ready(function() {
    $('#hello-button').on('click', function() {
        $.ajax({
            url: '/hello',
            method: 'GET',
            success: function(data, textStatus, jqXHR) {
                console.log(textStatus);
                alert('Hello ' + data);
            },
            error: function(jqXHR, textStatus, errorThrown) {
                alert('textStatus: ' + textStatus + ', errorThrown: ' + errorThro\
wn);
            }
        });
```

```
    });
});
```

(the above code snippet online)

With `$.ajax` in place, the difference now is that we have to specify the HTTP verb, using the `method` attribute.

Update the DOM

Usually, the successful completion of an Ajax request is being used to update the page, i.e. to update the DOM.

Let's use this to delete a row from the DOM on the page that lists all the articles. The row will be removed every time we delete an article. Hence, we will not have to update the whole page.

This is what we will actually implement:

Locale Is Set Based on Accept Language Header Value

Let's do it.

> Please, remove the `Hello` button from the application layout and remove the corresponding JavaScript code from the `app/assets/javascripts/application.js`. Also, remove the corresponding entry from the `config/routes.rb` file.

Change the Delete Action HTML Markup

First we need to change the way we generate the Delete action on the UI:

```
- # File: app/views/articles/_article.html.haml
- #
%tr.article-row-in-list-of-articles
  %td= article.id
  %td= article.title
  %td= article.text
  %td= article.user.email
  %td= l article.created_at, format: :short
  %td
    = link_to 'Show', article_path(article)
    - if current_user && current_user.can_write_article?(article)
        
      = link_to 'Edit', edit_article_path(article)
        
      = link_to 'Delete', article_path(article), class: 'btn btn-danger js-delete\
-article'
```

(the above code snippet online)

The changes are on lines 3 and 15. On line 3, we have added the class `article-row-in-list-of-articles`. This will be useful later on in our JavaScript code in order to locate the row that we need to delete. On line 15, we added the class `js-delete-article` and we removed the `method` attribute and the `data: {confirm: ...` data attribute. We don't need them because we will attach our own JavaScript that will handle both.

> *Hint:* Note that when we add classes and then we bind JavaScript events on them, we prefer to prefix their name with `js-` in order to show that these classes are being used for dynamic programming with JavaScript rather than for styling.

Bind JavaScript Handler

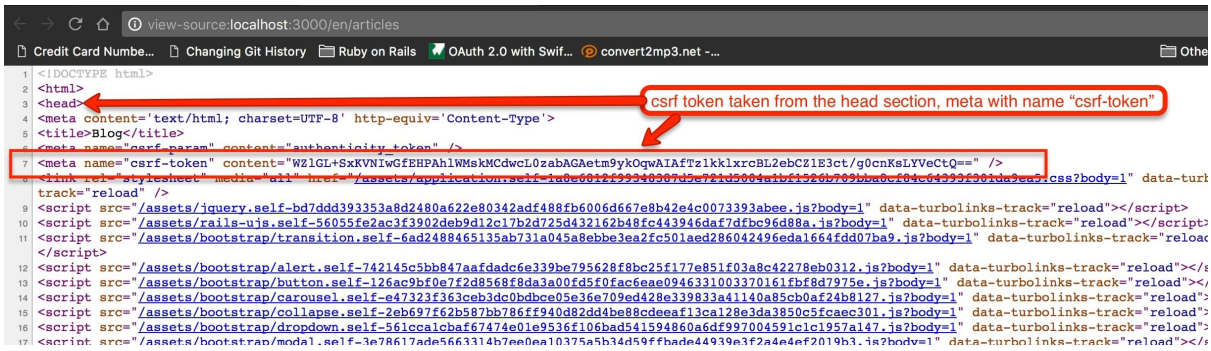
Create the file `app/assets/javascript/delete_article.js` with the following content:

```
// File: app/assets/javascripts/delete_article
//
$(document).ready(function(){
    $('.js-delete-article').on('click', function(event) {
        var $button = $(this);
        event.preventDefault();
        var answer = confirm('Are you sure that you want to delete this article?'\
);
        if (answer) {
            $.ajax({
                url: $button.attr('href'),
                method: 'DELETE',
                headers: {'X-CSRF-Token': $('meta[name="csrf-token"]').attr('cont\
ent')},
                success: function(data, textStatus, jqXHR) {
                    console.log('article ' + article_id + ' deleted!');
                    $button.
                      closest('.article-row-in-list-of-articles').
                      fadeOut(function() {
                        $(this).remove();
                      });
                },
                error: function(jqXHR, textStatus, errorThrown) {
                    alert('textStatus: ' + textStatus + ', errorThrown: ' + error\
Thrown);
                }
            });
        }
    });
});
```

(the above code snippet online)

This JavaScript code:

1. Binds a handler for the `click` event on the `.js-delete-article` elements.
2. The handler
    1. asks the user whether they really want to delete the article.
    2. If the answer is *yes*,
        1. it sends an Ajax request to the URL that is the value of the `href` attribute of the link that the user clicked on.
        2. The method is set to be `DELETE`, hence the HTTP verb is going to be `DELETE`.
        3. We also send the header `X-CSRF-TOKEN` having the value of the meta tag with name `csrf-token`. This is located at the `head` part of the page (see the screenshot below). If we don't do that, Rails will not accept the request. Remember that every time we make an HTTP request that might change the state at the server side (verbs: `POST`, `PUT`, `PATCH`, `DELETE`), then we need to include the CSRF token. This is done automatically with the `= form_tag` helper, but we need to do it manually with the `$.ajax()` requests.
        4. On success, we locate the row the button clicked belongs to, with the `$button.closest(...)` call. And then we fade out this row. At the end of the fade out, we remove it.



CSRF Token Taken From HTML head

Update Controller

But the above is not enough. We need to adapt our `ArticlesController#destroy` action. This is the new version:

```
# File: app/controllers/articles_controller.rb
#
class ArticlesController < ApplicationController
...
  end

  def destroy
    @article.destroy
    head :ok
  end

  def foo
...
end
```

(the above code snippet online)

The new `destroy` implementation is very simple. It does not return any body in the HTTP response. It only returns 200 OK header.

Now, everything is in place. Go ahead and test that your delete action works as expected.

Rails and Ajax

We have done the job to issue the delete request using Ajax. Now that we know how to do it, maybe it's about time to see what Rails can do for us in this context.

In short terms, Rails can provide us with helpers that allow us to write as less JavaScript code as possible. Sometimes without any JavaScript code. We will demonstrate that with a second delete button, `Delete 2`, so that you can have both implementations in parallel for comparison.

Ajax View Helper

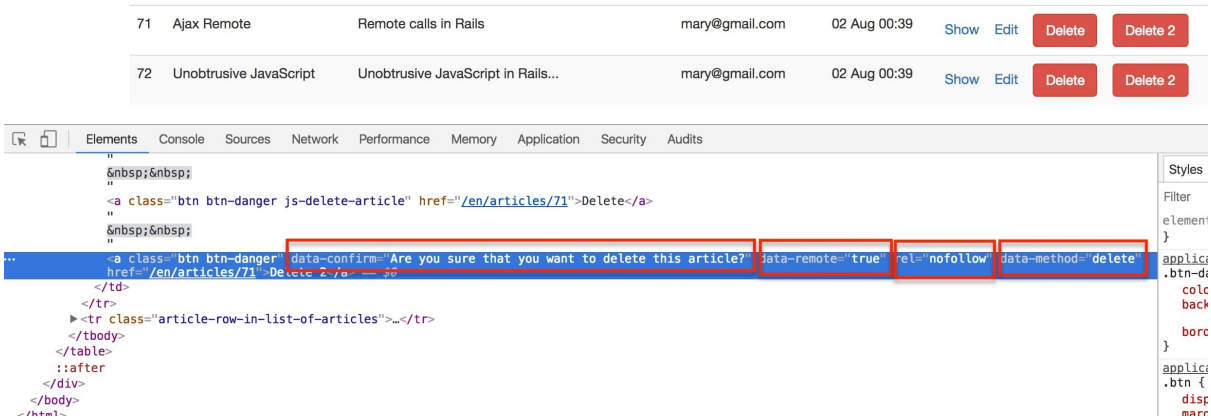
Let's change the view file `app/views/articles/_article.html.haml` so that it has another button next to the first one:

```
1  - # File: app/views/articles/_article.html.haml
2  - #
3  %tr.article-row-in-list-of-articles
4    %td= article.id
5    %td= article.title
6    %td= article.text
7    %td= article.user.email
8    %td= l article.created_at, format: :short
9    %td
10     = link_to 'Show', article_path(article)
11     - if current_user && current_user.can_write_article?(article)
12         
13       = link_to 'Edit', edit_article_path(article)
14         
15       = link_to 'Delete', article_path(article), class: 'btn btn-danger js-delete\
16 -article'
17         
18       = link_to 'Delete 2', article_path(article), class: 'btn btn-danger', remot\
19 e: true, method: :delete, data: {confirm: "Are you sure that you want to delete t\
20 his article?"}
```

(the above code snippet online)

Do you see line 17? Compare it to line 15. It does not have the custom class (`js-delete-article`) but it has a `remote: true` attribute set. Also, it needs the method specification, i.e. the HTTP verb we want to use for the link, since this is not a GET verb. Finally, note that we have added the confirmation question like we had it before, from the previous chapters.

Let's see the code that is generated with this `link_to` in place:



Anchor With Data Attribute remote true

Please, note the following:

1. The anchor has the `data-remote` attribute. This tells Rails that we want the click to be handled using Ajax call to server.
2. Also it has the `data-method` equal to `delete`. This is the HTTP verb, hence it's going to set the `method` attribute of the Ajax call.
3. The `data-confirm` has the question to ask for the user to confirm whether they want to proceed with the action on the anchor.
4. The `rel="nofollow"` tells crawling robots that this link should not be crawled.

Now, you can bind JavaScript function handlers to specific events, that have been designed for this particular reason. Here is what we write inside the `app/assets/javascripts/delete_article.js`:

```
// File: app/assets/javascripts/delete_article
//
$(document).ready(function(){
    $('.js-delete-article').on('click', function(event) {
...
    });

    $('.article-row-in-list-of-articles a[data-remote][data-method="delete"]').on\
('ajax:success', function() {
        var $button = $(this);
        $button.
          closest('.article-row-in-list-of-articles').
          fadeOut(function() {
            $(this).remove();
          });
    });
});
```

(the above code snippet online)

After the handler for the `click` event on the `.js-delete-article` selector, we have added a handler for the anchors that have been declared with `data-remote` and `data-method="delete"`. The handler handles the event `ajax:success` on these anchors. This event is attached by Rails JavaScript tools when we add the `remote: true`. If you see the implementation of the handler is the same like the `success` handler attached to the `$.ajax` call inside the `click` handler for the `.js-delete-article`.

Pretty cool, isn't it? We write much less code now. And we don't write code to send the Ajax request ourselves. It is executed behind the scenes by Rails JavaScript tools, only because we have added the `remote: true` attribute.

The events that you can bind JavaScript handlers are:

1. `ajax:before`, which is called before any Ajax code starts execution. If your handler returns `false`, then Ajax is aborted.
2. `ajax:beforeSend`, which is called before the Ajax request takes place. If your handler returns `false`, then Ajax is aborted.
3. `ajax:send`, which is called when the request is sent.
4. `ajax:success`, which is called whenever the response from the server is in the 200-399 range.
5. `ajax:error`, which is called whenever the response from the server is not in the success range.
6. `ajax:complete`, which is called at the end, in both success and error cases.

Other UI Elements with Ajax Support

Besides the `link_to` helper that supports Ajax calls, you can also use the `button_to`, to create buttons that they support Ajax. Also, you can use the helper `form_with` to create forms that will be submitted with an Ajax request.

If you want to read more about Rails and Ajax, you can read the Rails Guides here.

Action Cable

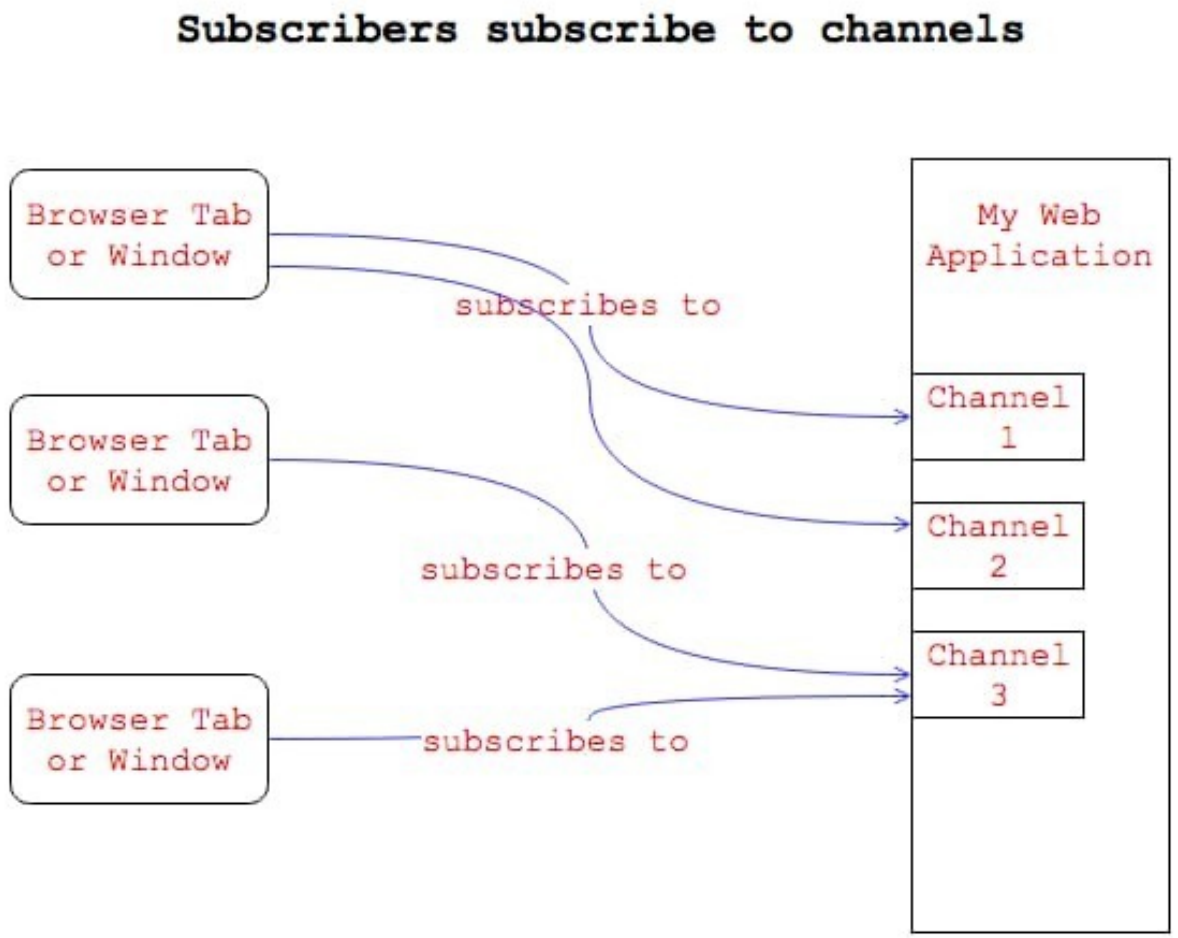
Having learned about the JavaScript support in Rails, we move on to Action Cable. Action Cable is the feature that Rails offers in order to provide real-time communication from the server back to the browser. The underlying technology is Web Sockets, but Action Cable makes it much easier to integrate it into your Rails application.

We have already studied how a browser uses HTTP to request data from the server. With WebSockets, we can have a TCP connection between browser and server that is full-duplex, which means that the server can send data to the browser without browser issuing a request. This technology can be used to push real-time information to users, when the information becomes available, without having browser polling the server periodically, asking whether new information is available or not.

Let's see how Action Cable helps us take advantage of WebSockets then.

Channels And Subscribers

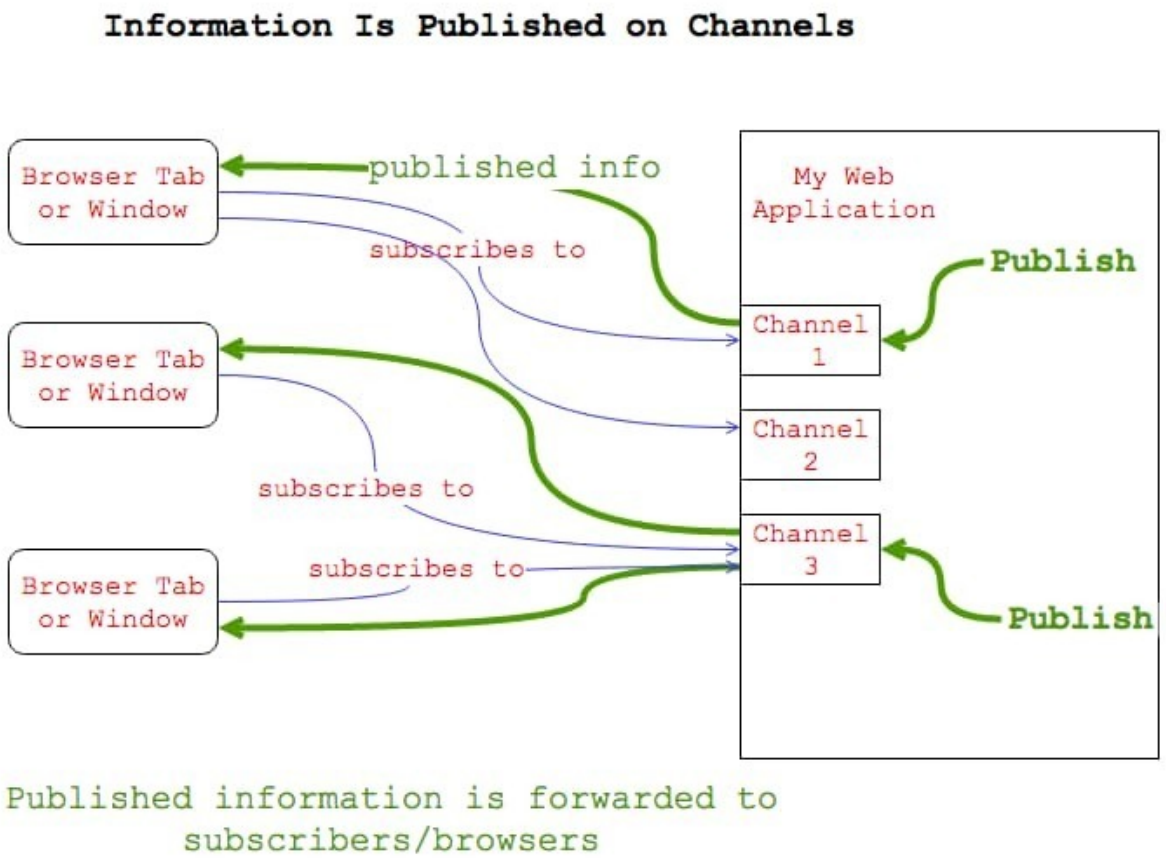
Rails allows you to define different *channels* and then have *subscribers* subscribing to any or all of these channels. The subscribers are the browsers and the channels are created at the server side:



Browser Subscribe To Channels

Publication Of Information

Then, your Web application, the server-side, publishes information to the channels, which is forwarded to all the subscribers that subscribe to these channels.



Subscribers Received Information That Is Published On Channels

Note that *subscribers* are initially called *consumers*, until they subscribe to a channel. Also, the fact that a consumer creates a WebSocket connection to the Rails server is represented by a *connection*, more precisely, by an `ApplicationCable::Connection`.

`ApplicationCable::Connection`

The `ApplicationCable::Connection` is a class already defined for us, inside our project folder:

```
# File: app/channels/application_cable/connection.rb
#
module ApplicationCable
  class Connection < ActionCable::Connection::Base
  end
end
```

(the above code snippet online)

It is empty, at the moment, and derives its behaviour from the `ActionCable::Connection::Base` class that is coming from the Rails framework. Every time a new connection is created, an instance of this class is being created too.

Also, if the instance responds to `#connect`, then this `#connect` method is called. Let's take advantage of it:

```ruby
# File: app/channels/application_cable/connection.rb
#
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    def connect
      logger.info "A new connection has been requested: #{self.connection_identif\
ier}"
    end
  end
end
```

(the above code snippet online)

We have implemented the method `#connect`. Hence, every time a connection is created, i.e. every time a consumer attaches to the WebSockets of our application via a subscription, we will see this message being printed in the Rails logs.

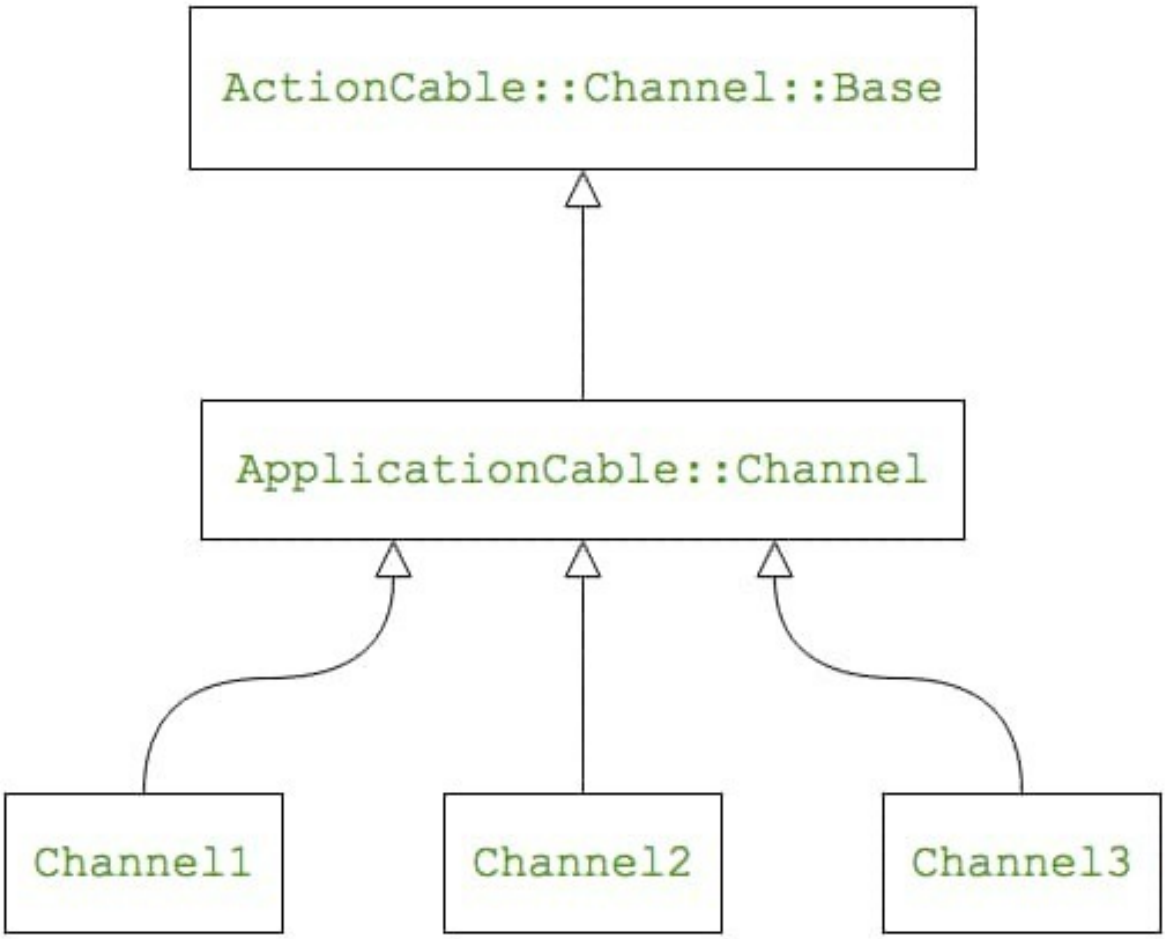
Channel Class

Your channels should be modeled as classes deriving from `ApplicationCable::Channel`. This class derives from `ActionCable::Channel::Base`. The `ApplicationCable::Channel` already exists. Look at the file `app/channels/application_cable/channel.rb`:

```ruby
# File: app/channels/application_cable/channel.rb
#
module ApplicationCable
  class Channel < ActionCable::Channel::Base
  end
end
```

(the above code snippet online)

Hence, your specific channels are going to be sub-classes of this class. Something like this:



Channel Class Hierarchy

The purpose of the existing `ApplicationCable::Channel` class is to put in whatever is going to be common to all your concrete channel classes. For example, let's just put a simple callback method, named `subscribed`. This is going to be called every time a subscriber connects/subscribes to a channel:

```
# File: app/channels/channel.rb
#
module ApplicationCable
  class Channel < ActionCable::Channel::Base
    def subscribed
      logger.info "A new subscription has been accepted: #{self.connection.connec\
tion_identifier}"
    end
  end
end
```

(the above code snippet online)

A Demo Channel

Create the class `ChatChannel` inside the file `app/channels/chat_channel.rb` as follows:

```ruby
# File: app/channels/chat_channel.rb
#
class ChatChannel < ApplicationCable::Channel
end
```

(the above code snippet online)

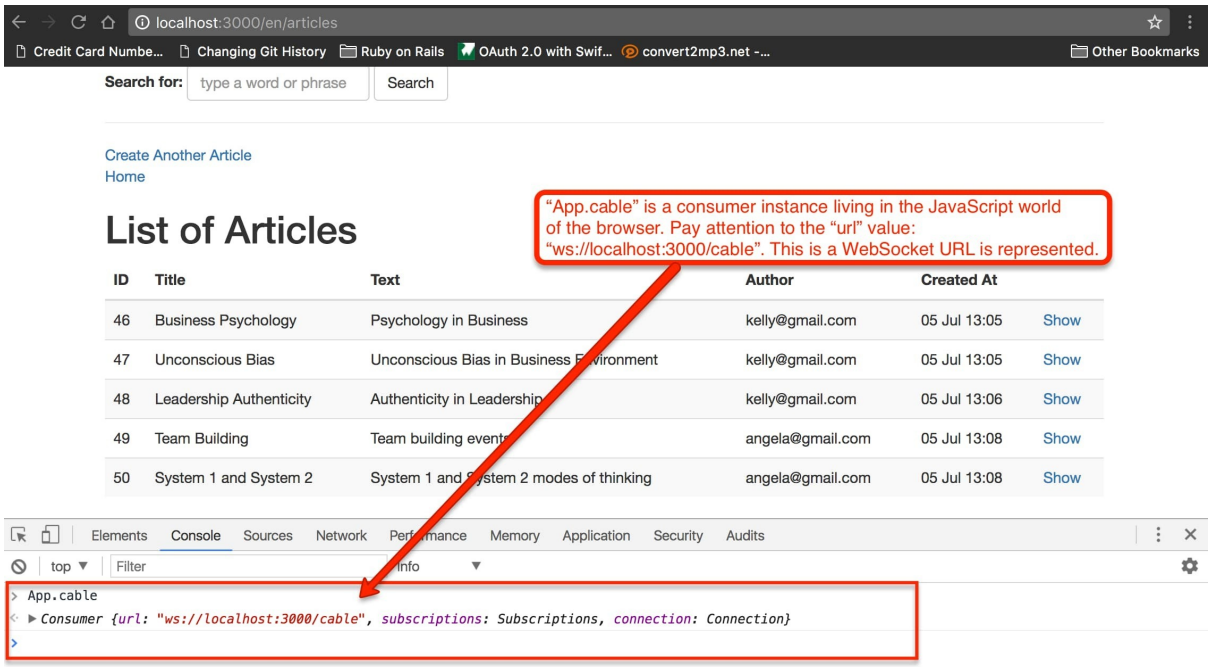
JavaScript Consumer

We may have the channel definition in the server-side, but we need some code at the browser, i.e. the consumer side too. Rails comes with the creation of a consumer instance saved inside `App.cable` property. This is done when your application starts, thanks to JavaScript code executed inside the `app/assets/javascripts/cable.js` file:

```javascript
// File: app/assets/javascripts/cable.js
//
// Action Cable provides the framework to deal with WebSockets in Rails.
// You can generate new channels where WebSocket features live using the `rails g\
enerate channel` command.
//
//= require action_cable
//= require_self
//= require_tree ./channels

(function() {
  this.App || (this.App = {});

  App.cable = ActionCable.createConsumer();

}).call(this);
```

(the above code snippet online)

The line `ActionCable.createConsumer()` creates a consumer at the browser side and this is saved inside `App.cable` property. You can confirm that by just visiting a page of your application and using the console to print the value of `App.cable`:



App.cable object holds a JavaScript Consumer instance

Do you see the `ws` URL? `ws://localhost:3000/cable`. This means that the consumer, if it decides to become a subscriber, it will connect at the `/cable` endpoint at your server side.

JavaScript Subscription

Having said that, creating a consumer is not enough. We need to make the consumer being a subscriber to a specific channel. This needs to be done with the corresponding JavaScript code.

Create the file `app/javascripts/cable/subscriptions/chat.js`. Note that the folder `cable/subscriptions` might not be present and, if not, you will have to create it.

```
// File: app/assets/javascripts/cable/subscriptions/chat.js
//
App.cable.subscriptions.create({
    channel: "ChatChannel"
}, {
    connected: function() {
        console.log('connected subscription for chat channel');
    }
});
```

(the above code snippet online)

The above code creates a subscriber that subscribes to channel represented by the class `ChatChannel`. Look how we define a callback function that is called when the consumer succeeds in making the connection to the `ChatChannel`. This callback is a JavaScript handler attached to the property `connected`.

However, the subscriber needs a little bit more code in order to handle the data published on this particular channel. We have not written this code yet. We will do it a little bit later.

Network and Rails Logs After Subscription

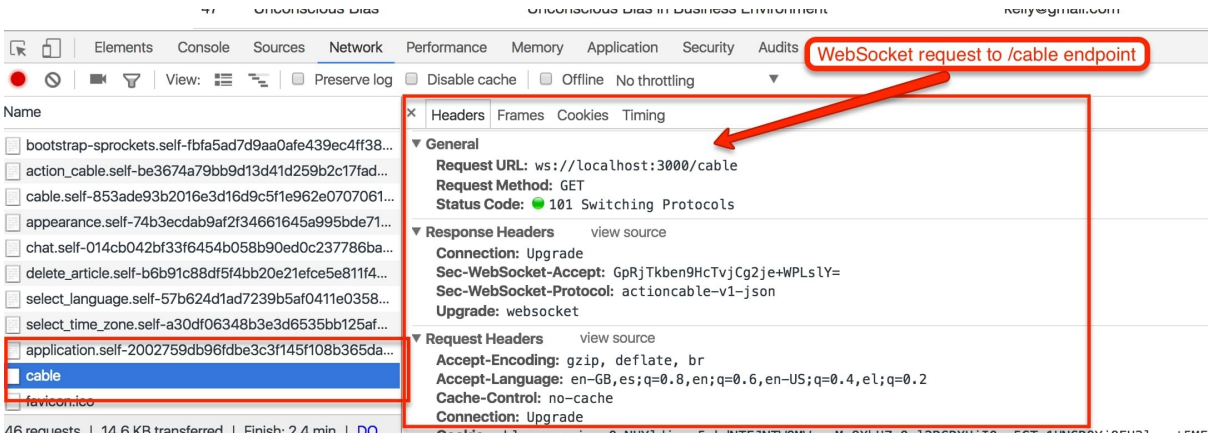
Everything is in place. Refresh your page and you will now start seeing Rails logs like these:

```
1  => Booting Puma
2  => Rails 5.1.1 application starting in development on http://localhost:3000
3  => Run `rails server -h` for more startup options
4  Puma starting in single mode...
5  * Version 3.8.2 (ruby 2.3.3-p222), codename: Sassy Salamander
6  * Min threads: 5, max threads: 5
7  * Environment: development
8  * Listening on tcp://0.0.0.0:3000
9  Use Ctrl-C to stop
10 Started GET "/cable" for 127.0.0.1 at 2017-08-05 13:47:32 +0300
11    (0.7ms)  SELECT "schema_migrations"."version" FROM "schema_migrations" ORDER B\
12 Y "schema_migrations"."version" ASC
13 Started GET "/cable/" [WebSocket] for 127.0.0.1 at 2017-08-05 13:47:32 +0300
14 Successfully upgraded to WebSocket (REQUEST_METHOD: GET, HTTP_CONNECTION: Upgrade\
15 , HTTP_UPGRADE: websocket)
16 A new connection has been requested:
17 A new subscription has been accepted:
18 ChatChannel is transmitting the subscription confirmation
```

(the above code snippet online)

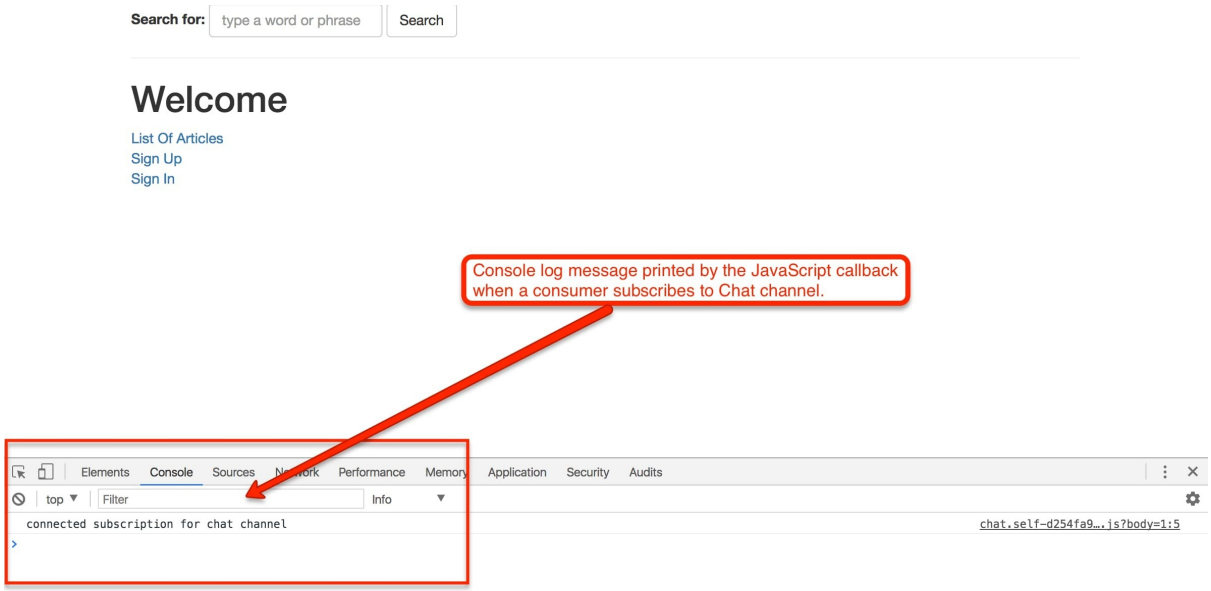
Do you see what happens when the `GET "/cable"` request is received? Look at lines 10 to 16. You can see the log lines that correspond to the connection and subscription logs that we have inserted earlier on.

Also, you will be able to see the WebSocket requests to the `/cable` endpoint if you open the Network tab in the Google Chrome developer tools:



WebSocket Connection To cable Endpoint

And if you open the Console tab, you will be able to see the log `connected subscription for chat channel` which is printed by the line 7 of `app/assets/javascripts/cable/subscriptions/chat.js`.



Console Log Message Confirming Successful Completion of Consumer Subscription

Before we leave the subject of logs, you might have spotted a problem. These logs that they are supposed to be printing the `connection_identifier` they don't print it. For example, the subscription log `A new subscription has been accepted:` does not end with the connection identifier. This is because we have not told Rails what makes a connection unique or identifiable. We will talk about it a little bit later.

Publishing Information

The connection and the subscription happen only once after the page has been loaded. Now the subscriber waits for data to be published on the `ChatChannel`, so that it can consume them.

Broadcasting Streams

In order for the server to be able to send data to the client, i.e. to the subscriber, it needs to set up a broadcasting stream. This can be done with the method `stream_from` that can be called at the channel level. Look at the following:

```ruby
# File: app/channels/chat_channel.rb
#
class ChatChannel < ApplicationCable::Channel
  def subscribed
    super
    stream_from 'chat'
  end
end
```

(the above code snippet online)

The `ChatChannel` class, for each new subscriber, it defines what is the broadcasting stream this subscriber is going to get data from. This is done on line 6. `stream_from 'chat'` means that the subscriber that has just subscribed is going to get data from the `chat` broadcasting stream.

Now, if you refresh your browser page, you will see one more line in the Rails logs:

```
ChatChannel is streaming from chat
```

(the above code snippet online)

Publishing

Now, we need to write some code that would publish data to the `chat` broadcasting stream (which is streamed out to subscribers by the `ChatChannel`).

One way that you can publish data to a broadcasting stream is with the following method:

```ruby
ActionCable.server.broadcast('chat', message: 'Hello World!')
```

(the above code snippet online)

The above code is broadcasting to `chat` stream the `Hash { message: 'Hello World!' }`. This is converted to `JSON` object and it is received in the JavaScript world by the subscribers that consume the Chat channel.

However, although you could try that on the rails console, it will not work. This is where we have to talk a little bit more about the *subscription adapters*.

Subscription Adapters

The subscription adapter is a piece of Rails code that allows the published messages to be forwarded to the registered subscribers. Behind the adapter is a tool that saves the message into a pool/database and then pushes the messages, asynchronously, to all the subscribers. Rails comes with the `async` subscription adapter when we are in `development` or `test` environment. However, it uses the `redis` subscription adapter when we deploy our application to `production` environment.

> *Information:* `Redis` is a NoSQL database and you can read more about it in the official Redis documentation page.

Which subscription adapter is used per environment is defined inside the file `config/cable.yml`. This is the content of the file in a brand new application:

```
# File: config/cable.yml
#
development:
  adapter: async

test:
  adapter: async

production:
  adapter: redis
  url: redis://localhost:6379/1
  channel_prefix: blog_production
```

(the above code snippet online)

> Reading the above content confirms what we have told you about which subscription adapter is used per environment. Please, note that your `production` setting assumes that your production server has Redis locally installed, which might not be the case. On Heroku, you will have to install an add-on like Heroku Redis in order to use this subscription adapter. Note that the `async` adapter is not production ready and should not be used in production.

`async` adapter

Back to the note that our publishing from the rails console wouldn't work, we need to tell you that `async` adapter does not work when the operating system process that publishes is not the same as the operating system that pushes the data to the subscribers. It only works within the same operating system process. This is the reason that publishing from the rails console, or from a background worker, will not work with the `async` adapter.

`Redis` adapter

Because of the problem that we have identified above for the `async` adapter and in order to make things simpler, we are now suggesting you to install Redis server in your development machine.

You can install Redis in your Mac OS X environment using `brew`. Try `brew install redis-server` or `brew install redis`. When you have it installed, you will confirm that Redis server is running:

```
blog $ ps -ef | grep 'redis'
  501   752     1   0 26Jul17 ??         1:32.19 /usr/local/opt/redis/bin/redis-s\
erver 127.0.0.1:6379
blog $
```

(the above code snippet online)

As you can see, the Redis server runs on interface `127.0.0.1`, i.e. on your local machine, listening for connection on the port `6379`.

With Redis server running in your local machine, change the content of the `config/cable.yml` file so that it does not use the `async` adapter in your `development` environment.

```
# File: config/cable.yml
#
development:
  url: redis://localhost:6379/1
  channel_prefix: blog_development

test:
  adapter: async

production:
  adapter: redis
  url: redis://localhost:6379/1
  channel_prefix: blog_production
```

(the above code snippet online)

And then, **make sure that you restart your Rails server** so that it takes into account the new settings.

Try Rails Console Publishing

With Redis subscription adapter, the Redis server installed and running and the Rails server restarted, refresh your web page.

Then, start your Rails console and give the following command:

```
blog $ bin/rails c
Loading development environment (Rails 5.1.1)
2.3.3 :001 > ActionCable.server.broadcast('chat', message: 'Hello World!')
[ActionCable] Broadcasting to chat: {:message=>"Hello World!"}
 => 1
```

(the above code snippet online)

Bingo! The return value `1` indicates that the JSON message `{"message": "Hello World"}` has been pushed successfully.

Subscriber Handling Broadcast Message

But how does the subscriber handles the broadcast message, i.e. the message that is published on the channel it subscribes to?

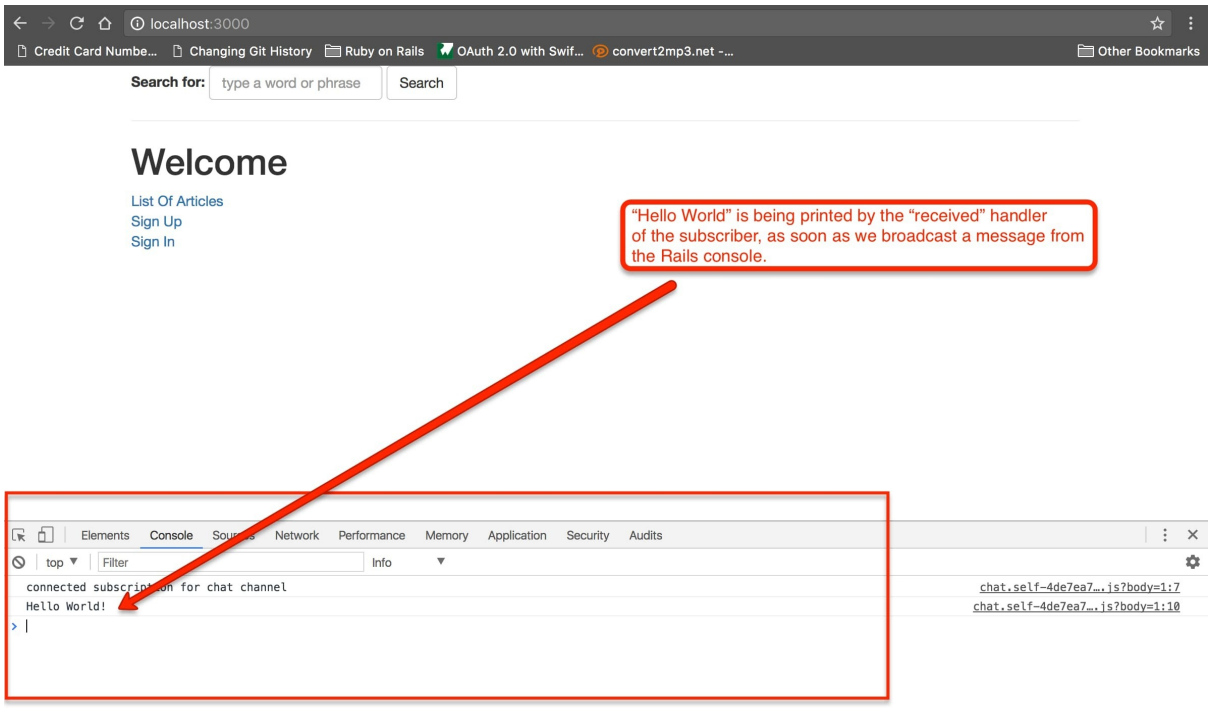
This is every easy. We need to implement the `received` callback at the JavaScript level for the particular subscriber. This is the new version of the file `app/assets/javascripts/cable/subscriptions/chat.js`:

```
1  // File: app/assets/javascripts/cable/subscriptions/chat.js
2  //
3  App.cable.subscriptions.create({
4      channel: "ChatChannel"
5  }, {
6      connected: function() {
7          console.log('connected subscription for chat channel');
8      },
9      received: function(data) {
10         console.log(data['message']);
11     }
12 });
```

(the above code snippet online)

Do you see lines 9 to 11. It is a JavaScript function handler that will be called when the subscriber receives data from the channel. Also, the actual data, will be a JSON object stored in the variable `data`. The JSON object will have the structure matching the structure for the data published in the Chat channel.

With the above in place, refresh your Web page and then send the publish message again from your Rails console. This is what you will see on the Console tab on your Web page:



Subscriber Prints Received Message

That is awesome! Our server pushes information to the subscribers of a channel!

Multiple Subscribers

The whole setup is much more exciting when we take advantage of the fact that the broadcast message is being received by all subscribers.

Try to open a new tab or a new window and visit your Web page. Then publish a new message from your Rails console. You will see that all the open tabs and windows that they have your page loaded will receive the message.

Publish To Multiple Subscribers

Also, you will see the corresponding log entries in the Rails logs. This is what I've got for the publications of messages that you saw on the above video:

```
ChatChannel transmitting {"message"=>"Hello World!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello World!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello World!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello World!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello Guys!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello Guys!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello Guys!"} (via streamed from chat)
ChatChannel transmitting {"message"=>"Hello Guys!"} (via streamed from chat)
```

(the above code snippet online)

Subscriber Publishes Message

But it is not only the server that can publish messages to a channel. It is the subscriber that can do that too.

First Save Reference To Channel

In order to achieve that, you first need to save a reference to the channel subscription created at the subscriber level, i.e. in the JavaScript create subscription code:

```
// File: app/assets/javascripts/cable/subscriptions/chat.js
//
App.chatChannel = App.cable.subscriptions.create({
...
});
```

(the above code snippet online)

The only difference is that we save the result of `App.cable.subscriptions.create()` to the `App.chatChannel` property of the `App` object.

Implement `receive` Method on Channel

We need to implement the method `receive` in the `ChatChannel` level:

```
1  # File: app/channels/chat_channel.rb
2  #
3  class ChatChannel < ApplicationCable::Channel
4    def subscribed
5      super
6      stream_from 'chat'
7    end
8
9    def receive(data)
10     ActionCable.server.broadcast('chat', data)
11   end
12 end
```

(the above code snippet online)

Do you see lines 9 to 11. The `receive` method will be responsible to handle the incoming messages from subscribers.

Try JavaScript Subscriber Publishing

With all the above in place, let's use some JavaScript code to publish a message from one of the subscribers. Open a series of a subscribers (new tabs or windows). Then on one of the subscribers Console tab, issue the following command:

```
App.chatChannel.send({message: "Hello From One Subscriber"});
```

(the above code snippet online)

You will see this message appearing on all subscribers.

Publish To Multiple Subscribers

Implement a UI To Send Messages

Having done all this preparation work, now it is easy for us to implement the following UX in which a subscriber sends messages to all other subscribers.

Sending And Displaying Received Messages

Let's do this implementation:

The HTML Markup

The HTML markup will be generated from a partial named `app/views/layouts/_send_-message.html.haml`:

```
- # File: app/views/layouts/_send_message.html.haml
- #
.form-group
  = form_tag do
    = label_tag :message_to_send, 'Message To Send Out:'
    = text_field_tag :message_to_send, nil, placeholder: 'Type in your message', \
class: 'form-control'
    = submit_tag 'Send Message', id: 'btn-send-message', class: 'btn btn-success'
  = label_tag :messages_received, 'Messages Received:'
  = text_area_tag :messages_received, nil, class: 'form-control'
```

(the above code snippet online)

This is a dummy form which does not actually submit its data to any endpoint. We will bind a click handler on its submit button and we will call the `App.chatChannel.send(...);` method to send a message to all the subscribers.

Sending Message

We now create the file `app/assets/javascripts/send_message.js` with the following content:

```
1  // File: app/assets/javascript/send_message.js
2  //
3  $(document).ready(function() {
4      $('#btn-send-message').on('click', function(event) {
5          event.preventDefault();
6
7          var $textWithMessage = $('#message_to_send');
8          var messageToSend = $textWithMessage.val().trim();
9          if (messageToSend !== '') {
10             App.chatChannel.send({message: messageToSend});
11             $textWithMessage.val('');
12         }
13     });
14 });
```

(the above code snippet online)

This is very simple implementation. The gist of it being line 10, on which we call the `App.chatChannel.send(...);`.

Receive Message

Chat channel receive message at the JavaScript level did only a `console.log` until now. Let's enhance it to append the message received into the text area:

```
1  // File: app/assets/javascripts/cable/subscriptions/chat.js
2  //
3  App.chatChannel = App.cable.subscriptions.create({
4      channel: "ChatChannel"
5  }, {
6      connected: function() {
7          console.log('connected subscription for chat channel');
8      },
9      received: function(data) {
10         var $messagesReceived = $('#messages_received');
11         var previousValue = $messagesReceived.val();
12         if (previousValue !== '')
13             previousValue = previousValue + '\n';
14         $messagesReceived.val(previousValue + data['message']);
15         $messagesReceived.scrollTop(
16             $messagesReceived.prop('scrollHeight') - $messagesReceived.height()
17         );
18     }
19 });
```

(the above code snippet online)

Have a look at lines 10 to 18. It is only a JavaScript code that appends the `data['message']` string into the text area that has the messages received. Appending the message at the end of the text area content was quite easy. Maybe, the most difficult part was to scroll the text area at the bottom. This is done with the help of .scrollTop() jQuery method.

So, now, everything is ready. Refresh the tabs with your Web application page and then start sending messages like in the video above. You will see all the messages being received by all subscribers.

Identification of Connections

The problem that we encounter with the above implementation is that the message sent is also received by the subscriber that has sent the message. This is something that we can remedy. We can change the code in order to make sure that the message is not pushed to the subscriber that has sent the message.

In order to achieve this, we have to identify, somehow, each connection. Here is the new connection code:

```
1  # File: app/channels/application_cable/connection.rb
2  #
3  module ApplicationCable
4    class Connection < ActionCable::Connection::Base
5      identified_by :connection_token
6
7      def connect
8        self.connection_token = SecureRandom.urlsafe_base64(36)
9        logger.add_tags connection_token
10       logger.info "A new connection has been requested: #{self.connection_identif\
11 ier}"
12     end
13   end
14 end
```

(the above code snippet online)

1. Firstly, we tell which attribute will uniquely identify each connection. This is done on line 5. We declare the attribute `connection_token` as being the one that would uniquely identify each connection.
2. Secondly, we have to give a unique value to the `connection_token` every time a new connection is created. This is done on line 8. We randomly create a string using `SecureRandom` library.
3. Note also how we add the `connection_token` value to the tags that will be used to tag the log lines.

Having done this, restart your rails server and watch the logs when you open tabs with your Web application pages:

```
Started GET "/cable" for 127.0.0.1 at 2017-08-06 12:57:45 +0300
Started GET "/cable/" [WebSocket] for 127.0.0.1 at 2017-08-06 12:57:45 +0300
Successfully upgraded to WebSocket (REQUEST_METHOD: GET, HTTP_CONNECTION: Upgrade\
, HTTP_UPGRADE: websocket)
[tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M] A new connection has been requ\
ested: tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M
[tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M] Registered connection (tpfrjxP\
5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M)
[tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M] A new subscription has been ac\
cepted: tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M
[tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M] ChatChannel is transmitting th\
e subscription confirmation
[tpfrjxP5yltXXq6BLM3Ow80DqLUvcgwZ-9AzUwG0GtgKfZ9M] ChatChannel is streaming from \
chat
```

(the above code snippet online)

Nice! Watch out also how the `self.connection_identifier` now prints the connection token value.

Fine Tune Broadcasting

Now, we need to tell the channel connection to transmit its message to a subscriber (note that we have one channel instance per subscriber) only if the connection identifier is not equal to the connection token of the channel connection:

```ruby
# File: app/channels/chat_channel.rb
#
class ChatChannel < ApplicationCable::Channel
  def subscribed
    super
    stream_from 'chat', coder: ActiveSupport::JSON do |message|
      unless message.with_indifferent_access[:connection_identifier] == connectio\
n.connection_identifier
        transmit message
      end
    end
  end

  def receive(data)
    ActionCable.server.broadcast('chat', data.merge({connection_identifier: conne\
ction.connection_identifier}))
  end
end
```

(the above code snippet online)

The first change is on the line 14, where we merge a `Hash` with the `connection_identifier` key having the value that uniquely identifies the connection for this particular `ChatChannel` instance.

Then, on the `stream_from` call, we do the other change. From a simple `stream_from 'chat'` call we now proceed to a more fine tuned implementation in which the `do ... end` block supplied has the logic to transmit the message or not. If you read the code, you will see that it calls `transmit message` only if the `connection_identifier` value of the `message` itself is different from the `connection_identifier` of the connection the channel belongs to.

Having done these changes, then sent messages are pushed to all subscribers except from the subscriber that actually sent the message.

Sending And Displaying Received Messages

Altering Message

Or you may want to send the message even to the sender itself, but with a prefix that would tell the recipient more about who was the sender:

```
1  # File: app/channels/chat_channel.rb
2  #
3  class ChatChannel < ApplicationCable::Channel
4    def subscribed
5      super
6      stream_from 'chat', coder: ActiveSupport::JSON do |message|
7        if message['connection_identifier'] == connection.connection_identifier
8          message['message'] = "You: #{message['message']}"
9        else
10         message['message'] = "Other: #{message['message']}"
11       end
12       transmit message
13     end
14   end
15
16   def receive(data)
17     ActionCable.server.broadcast('chat', data.merge({connection_identifier: conne\
18 ction.connection_identifier}))
19   end
20 end
```

(the above code snippet online)

The `stream_from do...end` block is now changed so that when we identify the sender to set the `You:` as the prefix in the message, whereas when we identify another subscriber we set the `Other:` .

Here is the video that demonstrate how it works:

Sending And Displaying Received Messages

Authentication and Authorization of connections

Before we close the subject of `ActionCable` just wanted to let you know that you may want to authenticate and authorize the incoming cable connections. Or even more, to identify the connection based on the current user logged in. Here is a sample connection code that would provide for it:

```ruby
# File: app/channels/application_cable/connection.rb
#
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      self.current_user = find_verified_user
      logger.add_tags current_user.name
    end

    def disconnect
      # Any cleanup work needed when the cable connection is cut.
    end

    private

    def find_verified_user
      User.find_by_identity(request.session[:user_id]) ||
          reject_unauthorized_connection
    end
  end
end
```

(the above code snippet online)

We will not elaborate more on this.

Closing on ActionCable

ActionCable is a fantastic tool but it needs your study and practice in order to master it. Read more about it in the Rails Guides or by googling it. There are a lot of articles out there on ActionCable.

Closing the Rails Section

That was our last chapter on Ruby on Rails. We believe that we have given you the necessary knowledge to work on Ruby on Rails as a junior Web developer. You will learn more if you practice and study more. The Rails Guides is something that you need to be going back frequently.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

You need to implement all the techniques presented in the chapter content for your blog application. This includes:

1. Hello button that prints the Hello World message on success using an alert. It prints error details on failure.
2. The `Delete` button in order to delete an Article. The button should work with custom Ajax code.
3. The `Delete 2` button that would work with the `remote` option and Rails Ajax support.
4. For `ActionCable` techniques, you need to implement the chat functionality as presented in the chapter

Important: Your code needs to be uploaded to your Github account.